RAY COLLECTION BOUNDING VOLUME HIERARCHY

by

KRIS KRISHNA RIVERA
B.S. University of Central Florida, 2007

A thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science
in the Department of Electrical Engineering & Computer Science
in the College of Engineering & Computer Science
at the University of Central Florida
Orlando, Florida

Fall Term
2011

Major Professor: Sumanta Pattanaik

ABSTRACT

This thesis presents Ray Collection BVH, an improvement over a current day Ray Tracing acceleration structure to both build and perform the steps necessary to efficiently render dynamic scenes. Bounding Volume Hierarchy (BVH) is a commonly used acceleration structure, which aides in rendering complex scenes in 3D space using Ray Tracing by breaking the scene of triangles into a simple hierarchical structure. The algorithm this thesis explores was developed in an attempt at accelerating the process of both constructing this structure, and also using it to render these complex scenes more efficiently.

The idea of using "ray collection" as a data structure was accidentally stumbled upon by the author in testing a theory he had for a class project. The overall scheme of the algorithm essentially collects a set of localized rays together and intersects them with subsequent levels of the BVH at each build step. In addition, only part of the acceleration structure is built on a per-Ray need basis. During this partial build, the Rays responsible for creating the scene are partially processed, also saving time on the overall procedure.

Ray tracing is a widely used technique for simple rendering from realistic images to making movies. Particularly, in the movie industry, the level of realism brought in to the animated movies through ray tracing is incredible. So any improvement brought to these algorithms to improve the speed of rendering would be considered useful and

welcome. This thesis makes contributions towards improving the overall speed of scene rendering, and hence may be considered as an important and useful contribution.

ACKNOWLEDGMENTS

I would like acknowledge the dedication, advice, and knowledge of my thesis advisor, Dr. Sumanta Pattanaik. Without him, I would not have been able to excel as I have and I appreciate all of the hard work, patience, and time that he has given me. Thank you so much for all you have done.

I would also like to thank my thesis committee members, Dr. Charles Hughes and Dr. Mark Heinrich. Both were my very supportive graduate professors and pushed me to want to learn and persevere as being a great student.

Finally, and most importantly, I would like to thank my family. They have supported me through every step and I am truly blessed to have them in my life. They are what I live for and, without them, would not be the person I am today. Thank you all so much.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ACRONYMS/ABBREVIATIONS

AABB	Axis Aligned Bounding Box
BVH	Bounding Volume Hierarchy
CPU	Central Processing Unit
GPU	Graphics Processing Unit
IDE	Integrated Development Environment
RCBVH	Ray Collection Bounding Volume Hierarchy
SAH	Surface Area Heuristic
USS	Uniform Spatial Subdivision

CHAPTER ONE: INTRODUCTION

Background

This research began with a school project at the University of Central Florida on Ray Tracing. The idea was conceived while attempting to fix an algorithm the author developed, which eventually yielded exceptionally good results. The algorithm presented attempts to aide a quite laborious process for rendering complex and realistic 3-D scenes using CPU based ray tracers. Ray Tracing is a technique wherein thousands to millions of Rays are cast out from a camera as to render objects within a scene. These rays are responsible for passing through a view plane and coloring the pixel through which they are traced. The color is determined by many factors, the main one being which object the ray hits closest to the view plane. After this 'hit', indirect rays are spawned, creating many different effects ranging from simple reflections and shadows to sub-surface scattering and translucency.

Though Ray Tracing is used for creating incredibly realistic scenes, it is also plagued by a very serious problem. It is a very compute intensive process and can take a very long time to be able to render even a small sized scene due to the nature of performing so many intersection tests and color calculations. The major problem comes from having to perform these tests for every triangle in the scene. Though a human observer can reason as to where a ray will hit or which object or triangle it will intersect, a computer has to be able to understand this in a different way. The worst process that can be performed to determine if a ray hits a certain triangle is the brute-force means of,

per each ray, looping over every triangle, determining which one the ray intersects. If it intersects many triangles, determining which triangle is closest is an additional (though very small) cost on top of the already expensive process. Fortunately, much work has gone into attempting to accelerate this slow process. One way of lessening the load of rendering these scenes is to create an acceleration structure. Of course, it takes time to build a structure, but the overall benefit of using them becomes apparent very quickly (especially as the number of triangles in a scene increases). With the aide of acceleration structures, the effort in building them faster and traversing them as efficiently as possible has become the focus of improvement. Rendering a dynamic scene poses additional problems. The general approach of re-building the acceleration structure for the scene is the simplest, but, as the brute-force method of Ray Tracing, this takes a brute-force approach for capturing the movement of triangles. In the recent years, developers have leveraged the use of sometimes hundreds of cores on the GPU to be able to create fast and real-time ray tracing algorithms. Again, the requirement to rebuild the acceleration structure per frame for dynamic scenes has severely restricted the wide use of GPUs. There have been some recent attempts in creating the acceleration structures in GPU. For example, Lauterbach et al. [4] leverage GPUs to perform the task of building BVH trees asynchronously in the GPU as to achieve fast ray tracing.

The goal of this thesis is to present a fast algorithm for performing Ray Tracing on a general purpose CPU. The thesis presents the author's attempt to alleviate the burden of creating an acceleration structure and rendering the scene separately by bringing

them together to exploit the inherent parallelism of Ray Tracing. The author intends to take the reader through a step-by-step process of the originating ideas which led to the actual discovery for the basis of this thesis as well as any valid attempt at deriving more speed from the algorithm.

Materials

The compute-platforms that are targeted in this work are not necessarily an overly powerful computer or cluster of them. All of the software that is used is open-source. The hardware used for implementing and testing the algorithms presented is an off the shelf commodity laptop. The details of the laptop are:

Model: HP dv6t

Processor: Intel Core i7 Quad-Core with Hyper-threading (8 threads of execution available)

RAM: 4GB of DDR2 667

HDD: 465GB 7200 RPM

Java was chosen as the platform for the thesis work, due to its wide adaptation and the ability for multi-threading. Eclipse, a widely used Java IDE, was chosen for the development of the Java code.

Language: Java

Java IDE: Eclipse for Java EE, Helios

CHAPTER TWO: LITERATURE REVIEW

The discussion in this chapter starts by outlining the basics of ray tracing, as a means of providing a sufficient algorithmic knowledge to the reader. Then it proceeds to describe acceleration structures and developments in that area. Then a brief background of the research performed in the field of ray tracing and acceleration structures is given. There exist algorithms for much faster ray tracing using GPUs, however, the main focus of this thesis is in using general purpose CPUs for faster ray tracing. If the reader feels that they do not require a detailed level of description for basics of ray tracing and acceleration structures, the reader may skip ahead to the research section in this chapter.

Ray Tracing Basics

The author feels that, to understand how the algorithm of this thesis works, it is important for the reader to understand the basics of Ray Tracing and BVH tree creation algorithms. Ray generation, indirect ray generation, ray-voxel intersection, and ray-triangle intersections are the necessary topics to understand.

Ray generation is the process by which rays are generated from a source observer and cast out through a view plane. To do this, the following user-defined variables are required: the position of the camera (**eye**), where the **eye** is looking (**at**), the vector deciding which direction is "**up**" in the scene, the distance the camera is from the view plane (**n**), and the dimensions of the view window (**width**, **height**). From these the vectors defining the camera's coordinate system can be calculated, where **u**, **v**, and

w represent the x, y, and z axis respectively. First, to calculate (**w**), which represents the unit vector opposite of the vector pointing from the eye position to the **at** position, the **at** position is subtracted from the **eye** and divided by the resulting vector's magnitude.

$$\boldsymbol{w} = \frac{\boldsymbol{eye}-\boldsymbol{at}}{||\boldsymbol{eye}-\boldsymbol{at}||} \tag{1}$$

The **u** vector, or the vector representing the 'x' axis of the coordinate system, can be calculated by taking the cross-product of the scene's **up** vector and **w** vector and, again, dividing the resulting vector by its magnitude.

$$\boldsymbol{u} = \frac{\boldsymbol{up} \times \boldsymbol{w}}{||\boldsymbol{up} \times \boldsymbol{w}||} \tag{2}$$

Finally, the remaining 'y' axis vector, **v**, can be calculated by taking the cross product of **w** and **u**. Since both **w** and **u** are already unit vectors, the resulting vector does not have to be divided by its magnitude.

$$\boldsymbol{v} = \boldsymbol{w} \times \boldsymbol{u} \tag{3}$$

The originating ray can now be calculated. To do this, a vector from the **eye** through the center of the first pixel in the view plane is created. A vector from the **eye** to the center of the camera is a simple multiplication of the distance from the view plane and the camera and **w**. Since **w** is pointing in the opposite direction, it has to be multiplied by negative one as well. This places the ray in the vertical and horizontal center of the view plane. For programmatic purposes, it is easier if the originating ray starts at a corner of

the view plane. To get to the bottom-left corner, the **u** vector is multiplied by negative the width of the view plane, divided by two. Similarly, the **v** vector is multiplied by negative the height and divided by two.

$$-\boldsymbol{w} * n - \boldsymbol{v} * \frac{height}{2} - \boldsymbol{u} * \frac{width}{2} \tag{4}$$

Finally, the size of a single pixel needs to be determined. If the pixel were 1x1, half of the pixel width and height have to be added to the above equation making the originating vector, **u00**.

$$\boldsymbol{u00} = -\boldsymbol{w} * \boldsymbol{n} - \boldsymbol{v} * \frac{\boldsymbol{height}}{\boldsymbol{2}} - \boldsymbol{u} * \frac{\boldsymbol{width}}{\boldsymbol{2}} + \frac{\boldsymbol{pixelWidth}}{\boldsymbol{2}} + \frac{\boldsymbol{pixelHeight}}{\boldsymbol{2}} \tag{5}$$

To generate the rest of the rays, **u00** is the base, then the outer loop goes by steps of the height of a pixel and the inner loop goes by steps of the width of a pixel until they reach the end of the width and height of the view plane, respectively. Please refer to Figure 1.

```
u00 = -w*n - v*height/2 - u*width/2 + pixelWidth/2 + pixelHeight/2
for (int i=0; i< vPixels; i++)
{
      for (int j=0; j< hPixels; j++)
      {
            rayDirection = u00 + u*pixelWidth*j + v*pixelHeight*i
            Ray(eye, rayDirection)
      }
}
```

Figure 1: Psuedo Code for Ray Generation

Next, the triangles which the ray may "hit" have to be found. This is performed as a two-step process. The first is to determine where, on the plane of the triangle, the ray intersects. If the ray is parallel to the plane of the triangle, then there will never be

an intersection. The simplest way to do this is by performing the dot product of the direction of the ray and the normal of the triangle. Since the cosine of the angle is part of the result of the dot product, if the angle between the direction of the ray and the normal of the triangle is either $90°$ or $270°$, then both the direction of the ray and the plane of the triangle are parallel with each other and will never intersect. Therefore for ray to intersect the triangle the dot product of the ray with the triangle normal should not be equal to zero.

$$\boldsymbol{n} \cdot \boldsymbol{d} \neq 0 \tag{6}$$

The point on the triangle's plane where the ray intersects it needs to now be calculated. The t range [t_min,t_max] of the ray-bounding volume intersection is computed by taking maximum of minimum t values to compute t_min, and minimum of maximum t value to compute to t_max. For a valid intersection t_min must be less than t_max. Then it must be divided by the dot product of the normal of the triangle and the direction of the ray.

$$t = \frac{(\boldsymbol{v} - \boldsymbol{Ray_{origin}}) \cdot \boldsymbol{n}}{\boldsymbol{n} \cdot \boldsymbol{d}} \tag{7}$$

If t is negative, it means that the ray *did* intersect the plane of the triangle, just in the opposite direction. That would not be useful, since the triangle would be behind the eye of the observer. The last step is to determine if this point is within the bounds of the triangle's vertices. For this, the barycentric coordinate of the point of intersection with respect to the triangle must be computed. Given a plane defined by two non-collinear

vectors (**v1**, **v2**) and passing through a point (p0), any point (p) on the plane can be specified two parameters, α (alpha) and β (beta), using the following equation.

$$p = p_0 + \alpha * vertex_1 + \beta * vertex_2 \tag{8}$$

If those variables – which represent how far along a given vector the point is – are within a certain bounds, then the point is known to be within the triangle. If those vectors are created, as follows, from the vertices of the triangle and the point of intersection then α and β can be computed, and be used to verify if the point is insider or outside the triangle by checking the sign and magnitude of α and β.

$$\boldsymbol{v} = p - vertex_1 \tag{9}$$

$$\boldsymbol{v_1} = vertex_2 - vertex_1 \tag{10}$$

$$\boldsymbol{v_2} = vertex_3 - vertex_1 \tag{11}$$

Notice how all of the vectors originate from the same point $vertex_1$ on the triangle. To find α and β for this point, the following operations have to be performed.

$$\alpha = \frac{(\boldsymbol{v_1} \times \boldsymbol{v_2}) \cdot (\boldsymbol{v} \times \boldsymbol{v_2})}{(\boldsymbol{v_1} \times \boldsymbol{v_2}) \cdot (\boldsymbol{v_1} \times \boldsymbol{v_2})} \tag{12}$$

$$\beta = \frac{(\boldsymbol{v_1} \times \boldsymbol{v_2}) \cdot (\boldsymbol{v_1} \times \boldsymbol{v})}{(\boldsymbol{v_1} \times \boldsymbol{v_2}) \cdot (\boldsymbol{v_1} \times \boldsymbol{v_2})} \tag{13}$$

For the point to lie inside the triangle, alpha and beta should be positive and less than equal to 1, and $\alpha + \beta$ should be less than equal to 1.The ray, most likely, will not intersect only one triangle, therefore, of the intersected triangles, it must be determined

which one is closest to the observer. This is simply done with a t value comparison. The value of t which is least amongst all of the triangles, which have been determined to intersect the ray, is the value which will be used to render the color of the pixel through which the ray is passing.

Another key area to consider is the indirect rays, which are rays that are generated off of a ray's intersection point with a triangle. They are used to compute, reflection, transparency etc... and hence make the scene far more realistic. There are many different types of indirect rays, but for the purposes of understanding in this thesis, only two types of rays will be considered: a reflective and a shadow ray. Once it has been determined that the ray originating at the camera (also known as primary rays) has hit a triangle, then these two indirect rays may be generated. First, and easiest, the shadow ray is calculated. Though many light sources and different type of them can be placed in a scene, here only one point light source will be considered, and the shadow ray can be calculated by subtracting the origin of the ray from the position of the light.

$$\boldsymbol{Ray}_{\boldsymbol{shadow}_{\boldsymbol{direction}}} = \boldsymbol{l} - \boldsymbol{p} \tag{14}$$

This ray is used to determine whether the intersection point is under shadow; in other words, if there is a triangle that is in the way along the direction towards the light source. If there is, the pixel is shaded to black or some other color of umbrage. The direction of the reflective ray is calculated by the following equation.

$$\boldsymbol{Ray}_{\boldsymbol{direction}_{\boldsymbol{in}}} - 2 * \boldsymbol{n} * (\boldsymbol{n} \cdot \boldsymbol{Ray}_{\boldsymbol{direction}_{\boldsymbol{in}}}) \quad (15)$$

Since the origin and the direction of the shadow and reflection rays (also called secondary rays) are known, it must finally be determined whether those rays intersect another triangle in the scene. The color of the triangle with which the reflective ray intersects is added to the pixel after modulating with the reflection coefficient of the triangle from which the ray originates. Pages 201 – 218 of [11] detail this entire section.

Acceleration Structures

In conjunction with the previous section, and of equal importance, a detailed description of acceleration structures is given here. The purpose of these structures is to accelerate the time-consuming process of ray tracing. Instead of having the rays traverse all triangles in the scene, the structures provide either a shortcut around having to render pixels or a faster path to get to the triangles with which they intersect. Axis Aligned Bounding Box (AABB), Uniform Spatial Subdivision (USS), and Bounding Volume structures are covered in great detail. The need for acceleration structures can be quite apparent when attempting to perform intersections

AABB

An Axis Aligned Bounding Box is a 3D box which contains the contents of an entire scene. The objective of AABB is to restrict unnecessary triangle traversal. Rays that do not intersect this bounding box do not proceed with the process of finding triangles with which they intersect. Similar to the a brute force method, if the ray does

intersect the bounding box, then all of the triangles within the box are looped over until all of the triangles are found.

To create the axis aligned bounding box, one has to loop over all of the triangles, recording maximum and minimum values along each given axis. If the value at a given vertex is greater or smaller than the current maximum or minimum, respectively, that becomes the new max or min. The following pseudo code describes the implementation for creating an AABB.

```
float[6] bounds; // [MinX, MaxX, MinY, MaxY, MinZ, MaxZ]
for (triangle in Triangles)
{
      for (vertex in triangle.vertices)
      {
            // Compare Min and Max X
            if (vertex.x < bounds[0])
                  bounds[0] = vertex.x
            if (vertex.x > bounds[1])
                  bounds[1] = vertex.x

            // Compare Min and Max Y
            if (vertex.y < voxelBounds[2])
                  voxelBounds[2] = vertex.y
            if (vertex.y > voxelBounds[3])
                  voxelBounds[3] = vertex.y

            // Compare Min and Max Z
            if (vertex.z < voxelBounds[4])
                  voxelBounds[4] = vertex.z
            if (vertex.z > voxelBounds[5])
                  voxelBounds[5] = vertex.z
      }
}
```

Figure 2: Pseudo Code for creating AABB

From there, ray-AABB intersections to determine whether the ray intersects that bounding volume are performed. This is done using a simple t-value test. The t-value is the constant multiplied by the direction vector of a ray and added to the origin to get a point in 3D space along the ray. So, by determining the values of t for the near and far

(x,y,z) planes of the bounding volume, it is determined if those values intersect the voxel's planes at 2 points that are within the range of the voxel's bounds. Any point on a ray can be found by adding its origin and the direction, multiplied by a value t.

$$p = \boldsymbol{Ray_{origin}} + \boldsymbol{Ray_{direction}} * t \quad (16)$$

The following equations can be used to calculate the t values intersection of the rays with the planes of the bounding volume perpendicular to the X-axis.

$$t_{min_x} = \frac{Voxel_{min_x} - Ray_{origin_x}}{Ray_{direction_x}} \quad (17)$$

$$t_{max_x} = \frac{Voxel_{max_x} - Ray_{origin_x}}{Ray_{direction_x}} \quad (18)$$

The t range [t_{min}, t_{max}] of the ray-bounding volume intersection is computed by taking maximum of minimum t values to compute t_{min}, and minimum of maximum t value to compute to t_{max}. For a valid intersection t_{min} must be less than t_{max}. This process can be found on pages 219-220 of Shirley et al.'s book [11].

USS

Though AABB limits the number of rays having to be processed, AABB has shortcomings in a couple of areas. When a ray intersects the bounding box, it still has to perform a brute force approach and loop through every triangle in it. The second area is that, if the bounds of the scene consume the entire viewable area, AABB operates just like the brute force method and provides no acceleration. One way to solve this is to perform Uniform Spatial Subdivision (USS). This is the process by which an AABB is

split into smaller boxes. The AABB is normally divided uniformly in n x n x n smaller boxes, also referred to as cells. Fortunately, since the bounds of the largest bounding box are already known, division into smaller boxes is simple. Depending on dimensions of the AABB, the width, height, and depth of each subdivided bounding box can be calculated.

$$w = \frac{AABB_{x_{max}} - AABB_{x_{min}}}{n} \tag{19}$$

$$h = \frac{AABB_{y_{max}} - AABB_{y_{min}}}{n} \tag{20}$$

$$d = \frac{AABB_{z_{max}} - AABB_{z_{min}}}{n} \tag{21}$$

From these dimensions, the smaller boxes can be built by performing triple nested loops going by the steps of w, h, and d. An (i, j, k) index should be assigned to each box, for easy traversal. Then, the triangles which belong to their respective cell need to be assigned. Each vertex of a triangle could be tested to see if it falls within the range of a small box.

Triangle assignment to the cells is done at the pre-processing step and the whole USS structure is accessed at ray tracing time. As with AABB, for every ray a ray-AABB intersection is made to ensure that the ray is going to at least hit one cell of the USS. Next, all the cells that are on the path of the ray are found. The direction of the ray has to be taken into consideration for this finding. If the ray's direction is negative, the search should begin, along that axis, on the maximum face of the AABB. The face of

the AABB that gets pierced first will determine which face of the cell to verify. Searching on that face, it may be found that more than one cell will be intersected, but the t-values of that intersection need to be compared to determine which one comes first. Since it is known that the triangles that are assigned to the cell under consideration, the nearest intersection is found, if any, among those triangles. Of the triangles within this box, the closest triangle is the one chosen. If the ray does not intersect any triangles, then the intersection test with the triangles in the next cell is carried out. The same process is repeated for all the cells on the path of the ray until a triangle is intersected with the ray, or the ray exits the back of the bounding box. This information can be referenced on pages 225-227 of [11].

One issue with this algorithm is if there is a particular box which contains a large portion of triangles in the scene, then, similar to the AABB, there are too many triangles that many rays have to traverse, and therefore, the algorithm becomes almost like the brute force method. If the triangles are well distributed through the scene, then this algorithm can be very good, but distribution cannot be guaranteed.

k-d Tree

Much like USS, k-d tree is space partitioning technique, used for partitioning points in 3D space. Short for k-dimensional, k-d trees differ from USS in that they are set up in a binary tree fashion. Starting at the AABB of the scene as the root node, each node is divided into a two nodes, by an axis aligned plane perpendicular to one of the axis. The point through which the plane passes may simply be the midpoint of the

extent of the node along that axis, or chosen using some clever heuristic. These planes of division provide the branches to the left and right sub-trees and partition the triangles of the parent node and place them on either side, and thus provide a spatial division.

k-d trees are an excellent acceleration structure because they are able to combine adaptivity to the primitive distribution in the scene, to the convenience of handling binary tree data structures. However, they carry a major shortcoming of USS in that, one cannot guarantee the assignment of the triangle primitives to a unique partition. It is possible that a triangle will intersect the partition plane and will be considered to be part of both the partitions. This complicates ray traversal and duplication of triangles adds to memory overhead. As will be see in the next section, BVH, the data structure central to this thesis, avoids these problems.

Bounding Volume Hierarchy

Building off of k-d trees is the idea of BVH structures. A BVH uses the same process for building a bounding volume as an AABB, the volume is split along a determined axis, and the two new volumes are placed as the children of the parent. Where the split is determined by the heuristics used by the developer, but in a general case, the axis along which the bounding volume is longest is chosen. Generally, where the split occurs is determined by an algorithm called Surface Area Heuristic (SAH). SAH is used for creating hierarchies which are efficient to ray trace quickly. Wald outlines how to perform a SAH in [10]. As a major variation to the k-d tree concept, the split is found and the triangles within the parent volume need to be placed to left or right of the

split point. The process of partitioning and AABB creation for the partition continues until the hierarchy is built where every leaf node has only one triangle within it.

From this point, it is assumed that the ray-scene intersection begins at the root node of the scene BVH structure. Each ray performs a ray-volume intersection test (which is the same as the ray-AABB intersection). If the ray intersects an intermediate node volume, the decision to traverse the left child node or right child node or both is taken by intersecting the ray with both nodes. Once the ray reaches a bounding volume with only one triangle, then the triangle is stored for later ray-triangle intersections. Once the ray is processed through the entire structure, the ray has all of the triangles with which it might intersect and, as detailed in the ray tracing basics section, the ray can get the appropriate color for the intersection. The process is detailed on pages 222-225 in [11].

Research

Acceleration structures, particularly the binary tree based ones, significantly improve the speed of ray tracing. Once the structure is created, fast browsing of the scene, sometimes at an interactive level, is possible. The initial cost incurred in creating the acceleration structure is amortized over multiple frames of rendering and hence is well justified. However, rendering of dynamic scenes, where objects are moving, requires repeated computation of such a structure for each rendered image frame. In this case the cost of rendering a frame includes the cost of the creation of the acceleration structure, and depending on the size and type of the scene, this latter cost

can become dominant. Therefore, current research on ray tracing is mostly devoted to faster dynamic creation of acceleration structures. Of most interest to this thesis is the research directed towards general purpose or commodity CPUs. Ize, Wald, and Parker [2] provide means of asynchronous BVH construction of dynamic scenes. They perform this on a “8 processor dual-core Opteron 880 shared-memory PC”, and use all 16 cores for their experiments. They use a single thread to asynchronously create the BVH structure while the remaining threads are used for performing rendering and BVH restructuring or refitting. Though building a BVH structure usually takes more than one frame to create, the other threads are kept busy refitting the previous tree built, updating vertices, and rendering the scene. They ensure to lock the process at different steps so that all of the threads are running in parallel, but are synchronized in the tasks that are being performed. The only flaw that the author finds in this approach is that they state that they reserve only a single core on an N-core machine for building the BVH tree. If they were to possibly use more than one, they may be able to find a ratio of cores for building the tree to the number of cores for performing the remaining tasks which would be even more favorable.

Continuing with the discussion of the research performed for ray tracing performed on general purpose CPUs, in [10], Wald uses SAH to build bounding volume hierarchies for fast ray tracing. In this implementation, Wald uses a k-d tree for BVH creation. He creates different cost equations to attempt to optimize the tree creation such that combinations of maximum tree depth, minimum cell size, and comparing the cost of splitting versus not splitting the node are all minimized. The cost of not splitting is

determined by number of rays passing through an un-split volume, the average time to traverse a cell in a kd-tree, the number of triangles in this cell, and the average time to perform a ray-triangle intersection. The cost of splitting depends, similarly, on the same variables, but instead, they are added up with the triangles in the splits. Depending on how far down the tree has been created, splitting of the nodes will stop since the cost of not splitting the node will cost less than the cost of splitting it. From this, Wald was able to come up with an equation detailing the time it takes to render a frame, based off of the number of traversed cells, the average time it takes to traverse a cell, the number of intersection tests, and the time to perform an intersection. After performing several hundreds of experiments, he was able to determine those average times for triangle intersection and cell traversal and plug them into his equations to be able to achieve supposed optimal times to build and render a scene. In addition to efficiency of the structure building and ray tracing, he took into account not only the amount of memory that was used, but also the locality of recently accessed objects.

Much of the recent work seems to leverage the use of GPUs for either fast or real time rendering. Though these algorithms are not completely applicable to the purpose of this thesis, the distinction between CPU and GPU capabilities seem to be slowly reducing, and hence algorithms designed for GPU may provide some insight into developing better algorithms for CPU and vice versa.. Pantaleoni and Luebke [6] created the Hierarchical LBVH (HLBVH) by extending the LBVH concepts which Lauterbach et. al [4] presented in their the Linear Bounding Volume Hierarchy (LBVH) algorithm. [4] Applies a hierarchical grid to take advantage of spatial coherence in the

input set. They first use a linear ordering which builds BVHs quickly, followed by “top-down” approach which uses Surface Area Heuristic to optimize them for fast ray tracing. They combine these methods into a hybrid approach which attempts to remove the bottlenecks in GPU. Making the problem of BVH construction into a sorting problem, they order all of the input primitives on a “space-filling” curve, then split the intervals recursively. They then bring an SAH to make a hybrid BVH builder in the GPU. The power of this approach is to take advantage of the top down SAH hierarchy builder while also removing some of its inherent bottlenecks by first using the LBVH algorithm to generate a shallow tree that is further refined in parallel using the breadth-first SAH construction algorithm. In [6], they are able to extend this to create a hierarchical LBVH for real time ray tracing. They take the original approach in [4], and split it into a 2-level hierarchical problem where the first level consists of sorting the primitives into a coarse grid to take advantage of spatial coherence in the input set. Once that tree is built, they continue with sorting and creating the missing levels of the BVH tree. With this, they are able to perform 2 to 3 times better than the algorithm presented in [4] and consume 10 to 20 times less bandwidth. Beyond the GPU, Wald has investigated Intel’s Many Integrated Core (MIC) architecture for the task of fast SAH BVH construction [8]. He showed significant increases on a highly parallel shared memory system.

The approach proposed by Wald, Ize and Parker [5] attempts to capitalize on the recent advances in multi-core processors. They provide an implementation for asynchronous BVH construction with a subset of available threads while the rest of the threads are dedicated to the rendering task, simply switching over to the new tree once

construction is completed. They attempt to hide the time it takes to build the structure during other processes. This approach prevents the system from stalling when new trees are required, while still generating new BVH trees before the current tree is too deformed making it very well suited for use on dynamic scenes. It uses a two stage system for BVH construction in order to remove apparent bottlenecks with normal top-down tree building. It proposes a coarse grained partitioning scheme and then parallel BVH sub-tree construction on the individual partitions. The power comes from the fact that if the scene can be partitioned into a grid of tree subsets, then each tree of the grid can be processed concurrently.

In addition to BVH usage for ray tracing systems, work has also been done in the realm of continuous collision detection (CCD). This field uses BVH trees as a way to perform collision detection between other objects as well as detecting inter-collisions between the mesh of a single object. In [9], the authors propose a breadth-first traversal method for handling collision detection test pairs. They split the tree up into high-level and low-level nodes. The high-level nodes represent the upper parts of the tree which are the least impacted by minor changes in geometry. Low-level nodes are the part of the construction front where the number of nodes is equal to the number of construction threads. At this point, each thread takes a low-level node as the root of a sub-tree and is able to process these sub-trees concurrently without the use of locks.

It is clear to see that there have been many advances in Ray Tracing focusing around the building of acceleration structures and how to perform these processes as

quickly as possible. GPU implementations are excellent, and not to take away from them, but many do not have access to heavy hitting hardware like this and therefore rely on the CPU to do the work. As CPUs continue to become more powerful, many of these GPU implementations can be used in the CPU and still perform well doing so, which is why the author provided research in both areas of CPU and GPU graphics research.

CHAPTER THREE: METHODOLOGY

This chapter describes the main contribution of this thesis. The chapter starts with the description of the original idea that led to the exploration of a number of different approaches and iterations, until the author finally refined them into the main idea of the thesis of Ray Collection BHV algorithm. The author would like to emphasize that the actual contribution of this thesis did not derive directly from any present day algorithms, but was derived through a number of original thoughts and trial and error. The chapter begins with the findings from the very first idea that the author conceived for a class project. Subsequently, the idea behind the thesis is explained and developed into the iterative algorithms, which are attempts at achieving a speedup on that algorithm. The use of multi-threading is employed as a means to do this, in a later implementation of the thesis algorithm as well.

Surface Area Heuristic Optimization

After having originally learned the theory behind Bounding Volume Hierarchies (BVH), the author decided that his class project would focus around a mechanism of optimization similar to how SAH is used as to make the process of Ray Tracing faster. SAH is designed to limit the cost of traversing the BVH and ensuring that the traversal of each ray down the hierarchy to its final set of triangles will be performed as quickly as possible. The main motivation behind the project was that since an SAH was based on just calculating the surface area of the partitioned bounding volume and relating that to its parent bounding volume's surface area, then one would be able to make the heuristic

far more accurate by having the exact knowledge of each ray to calculate which side of the partitioned bounding volume would be intersected. From that, one would then be able to determine which dimension of the bounding volume to partition for a far better ray intersection performance, and hence a better heuristic. Since one axis of the parent volume has to be chosen to split into two children, using this idea one would be able to choose the splitting that would benefit the ray the most.

Implementation

The SAH algorithm and any improvement to it, is invoked when a BVH (and any similar data structure) node that represents the bounding volume of a bunch of primitives, must be split to create two children. The first thing that needs to be done is that, for the volume in question, an axis needs to be chosen for splitting it into two subsections. Traditionally, the axis along which the voxel has the longest length is the one that is chosen. After choosing the axis, a point is chosen along it to split the parent bounding volumes into two children volumes. Since, merely from the primitives, one cannot guess the optimal place to perform a split is located – particularly because the primitive distribution is not normally represented in a fashion that one could easily query – a number of different fixed points are used. Generally, for algorithmic convenience, a fixed number, say 12, splits are chosen, the simplest choice being the one in which the split points are evenly distributed. For each split, two subsets of triangles are created: one with the triangles to the left of the split and the other with the triangles to the right of the split. Once all of the triangles are split into two lists, bounding volumes are created for the “left” and “right” triangle subsets. Then, the surface area of each of the volumes

is calculated and divided by the surface area of its parent volume, and finally multiplied by the number of triangles within the subset of triangles.

$$SAH_{cost} = \frac{SA_L}{SA_{parent}} * Triangles_L + \frac{SA_R}{SA_{parent}} * Triangles_R \quad (22)$$

The above equation gives us the cost of traversing any branch of a BVH, given the inputs. The cost for the current split point is compared with the cost of previous split point, and if the cost of the Surface Area Heuristic is smaller than the last one – i.e. the cost of traversal is less – then the "left" and "right" triangles split at their relative split value are saved for later use. This process is continued until all the split points are verified and the final result is used for the actual split of the bounding volume.

Results

The advantages of SAH based splitting is well documented. Wald documents these results in using SAH to build BVH's in [13]. The proposed improvement in the form of choosing the dimension based on individual ray was not fully implemented due to the fact that early on during the implementation, it became clear that there was a fatal flaw in the idea that was completely overlooked, hence the full implementation was abandoned. Even though the implementation was abandoned midway, and there was no tangible result, the exercise itself gave rise to many more ideas.

Discussion

Unfortunately, as it goes with most trial and error work, early on during implementation a major flaw in the idea became evident. The flaw was that thousands

of rays intersect the same voxel and therefore the decision of whether to split the parent node on a given axis based on a single ray would not work at all. There is no single representative ray that could determine the best axis to split the bounding volume since, especially at the highest level of the BVH, the volume is very large. This idea could possibly be used for smaller volumes, where the locality of the rays are almost all the same, but the problem with this is that there are thousands of indirect rays which would intersect the volume from different directions. So, the final conclusion was that this idea itself did not have the required merits, though there was one part of it that could have some importance. Since this SAH optimization was based around the idea of optimizing a part of an existing algorithm based on a single ray, this idea was used to fuel the next: Per Ray Path Creation.

Per Ray Path Creation

The lessons learned from the unsuccessful Surface Area Heuristic Optimization, led to the idea that it may be worthwhile focusing on building paths only for rays. So, instead of building the entire tree first, one would build only the paths necessary for each ray. This started off as a novel idea and seemed to have excellent potential. The reasoning behind this optimism was that only the paths down the BVH tree would get created as needed, therefore would achieve an overall speedup on the entire process of not only building the BVH tree, but also on the ray traversal and ray-voxel (bounding volume) intersection. One would only have to build the children of the voxels with which the ray intersected, and repeat this process recursively until the single ray had built all of the necessary paths and determined which set of triangles it might intersect. After it

was done, it would then discard those paths and continue to render the pixel it was responsible for coloring. The author felt that it would be simple to regenerate the paths that each ray required, therefore saving on memory.

Implementation

To implement this, the simple mechanism of bounding box creation around each triangle was sufficient; looping through each triangle and each vertex and storing the next largest and smallest 3D position (x,y,z). From this, a rectangular bounding volume which contains an entire set of triangles is built.

The next step in the process would normally be ray generation such that they could intersect the volumes and traverse the tree until all of the triangles with which the ray intersects is stored. For each ray, it checks whether it intersects with the BVH node in question. If it does intersect, the process of building children nodes is invoked. After this has completed, a recursive call is made to process the left child and right child nodes. If the ray does not intersect the node, then the recursion ends for that branch of the binary tree. If, after intersecting the BVH node, there is only one triangle stored for that node, then that triangle is assigned to the ray for later processing. The pseudo code for this algorithm is provided below.

```
function recursiveTreeBuild(ray, node)
{
      if (ray.intersects(node) ==  true)
      {
            if (BVH node.triangles.size == 1)
            {
                  ray.triangles.add(node.triangles)
                  return
            }
```

```
                BVH node.buildChildren()

                recursiveTreeBuild(ray, BVH node.leftChild)
                recursiveTreeBuild(ray, BVH node.rightChild)
        }
}
```

Figure 3: Pseudo Code for Per Ray Path Creation

This process is repeated for each ray in the scene. Due to the size that BVH trees can grow and in an attempt to save memory, the structure is not stored. Since the nodes store not only the bounding volume but also references to all the triangles contained within the volume, the sum of volumes can consume a large amount of memory.

Results

This idea was implemented completely and, unfortunately, the results were not very encouraging. From the results it was clear that algorithm missed out on something very crucial to its success. Where the normal implementation for building a BVH tree succeeded, the current algorithm hanged. In some cases, when the size of the scene grew too large, the execution took too long to even wait for. What should have been an efficient solution was not for reasons the author did not understand at the time. Due to the results of the experiments, the author did not record any times for them. From the results, it was clear that the algorithm was deficient in a certain area and required some rework.

Discussion

Though the results were not every encouraging, from a careful analysis of the results and of the algorithm, the source of the problem became clear. The problem is as follows: every ray entering the parent bounding volume would create its children. It

would only create the paths that were necessary to determine which triangles intersected the ray, but it would carry out this process over and over again. The biggest bottleneck was the parent bounding volume, which contained the entire set of triangles in the scene. So, for every single ray, the algorithm was iterating over all of the triangles and, for a small scene (800x600), there are still 480,000 rays intersecting it. And as the amount of the triangles in the scene grows, the amount of time that this method takes to complete is exponential. Essentially, the algorithm was trying to create a small part of the tree, but a significant number of times. Through subsequent analysis to correct the problem, the path to the foundation of the thesis was finally discovered: Ray Collection BVH.

Ray Collection BVH (RCBVH)

After identifying that the main issue that the previous algorithm was preforming so many unnecessary loops, the idea of single ray path traversal was brought together into one large collection of rays in the scene. The process, like the previous implementation, would start at the parent bounding volume, but instead of only having the single ray, the method would have access to the entire set of pre-constructed rays. Each ray would be intersected with the bounding volume and, if there was at least one intersection, then children for the volume would get created. This process would be repeated until a sparse tree was built. This tree would be sparse only because the nodes of the BVH tree which are needed get created. Since all triangles in a scene may not be present within view-volume starting at the near plane, but present in the data structure storing all of the triangles, this method would not bother creating the whole

structure. This partially built structure can then be used to find the triangles with which indirect rays intersect. If the paths of the tree are already built, then all of the indirect rays have to traverse the already built paths. But if the paths do not exist, then they get built as they are needed.

Implementation

In this implementation, most of the ideas from the previous implementation remain the same. First, before the BVH structure is built, all of the primary rays in the scene need to be generated. This is a relatively fast process, but is a crucial step for this algorithm. Normally this would be done one at a time and each ray would traverse the already built acceleration structure, but having collected all of the rays together, this process can be done all at the same time. After this collection of rays is created, the parent BVH node needs to be created. Once this is completed, then the BVH node and the collection of ray objects can be passed into the recursive method which will be responsible for verifying intersections and building children. In this method, a check is made to verify whether there is only one triangle within the BVH node. If this is true, then this triangle can be assigned to the collection of rays currently within the level of recursion. If there is more than one triangle within the BVH node, then intersection tests need to be performed as to determine which rays intersect the BVH node's bounding volume. For the rays that pass the intersection, they are stored in a separate collection. At least one ray which intersects the bounding volume has to exist to proceed, otherwise the recursive method exits. If there is at least one ray that has intersected the volume, then the parent node's children must be created (using the Surface Area

Heuristic method shown above). After the children are created recursive calls to the same method are made, with respect to the node's children, until the execution completes. After the primary rays are process, the indirect rays are able to be generated and passed through the same process. The following pseudo code depicts what is done for this algorithm (For the actual implementation of the algorithm, the reader is referred to appendix – getRCBVHTrianglesSynchronous()).

```
function recursiveTreeBuild(rays, node)
{
	ray[] intersectedRays
	for (ray in rays)
	{
		if (node.triangles.size == 1)
		{
			ray.triangles.add(node.triangles)
		}
		else if(ray.intersect(node)
		{
			intersectedRays.add(ray)
		}

	}

	if (BVH node.triangles.size == 1)
	{
		return
	}

	if (intersectedRays.size > 0)
	{
		node.buildChildren()

		recursiveTreeBuild(intersectedRays, node.leftChild)
		recursiveTreeBuild(intersectedRays, node.rightChild)
	}
}
```

Figure 4: Psuedo Code for Ray Collection BVH

The recursive method is guaranteed to exit at the appropriate times whether there is only one triangle in the node, or if there aren't any rays which intersected the bounding volume.

Of additional importance are indirect rays. At this point, after the primary rays have been processed, and a partially built bounding volume structure has been built. The indirect rays need to be generated and put through the same process. The primary rays have already generated most of the required paths necessary for the indirect rays, but if the path for an indirect ray is not present, it will simply generate the children recursively until the path is determined.

Results

The results of this algorithm are presented as follows. The timings encapsulate the pre-generation of all of the rays, the construction of the RCBVH tree, and the rendering of the rays afterward. For every model rendered, the camera begins at a farthest corner of the bounding volume which contains the entire model, therefore allowing the camera to see it entirely. Before the results are discussed, the author would like to explain the “curve-fitting” of the following graphs. The results do not have a mathematical curve fit applied to them but, instead, a simple smoothed connection from point to point as to visualize the comparative results of the different algorithms.

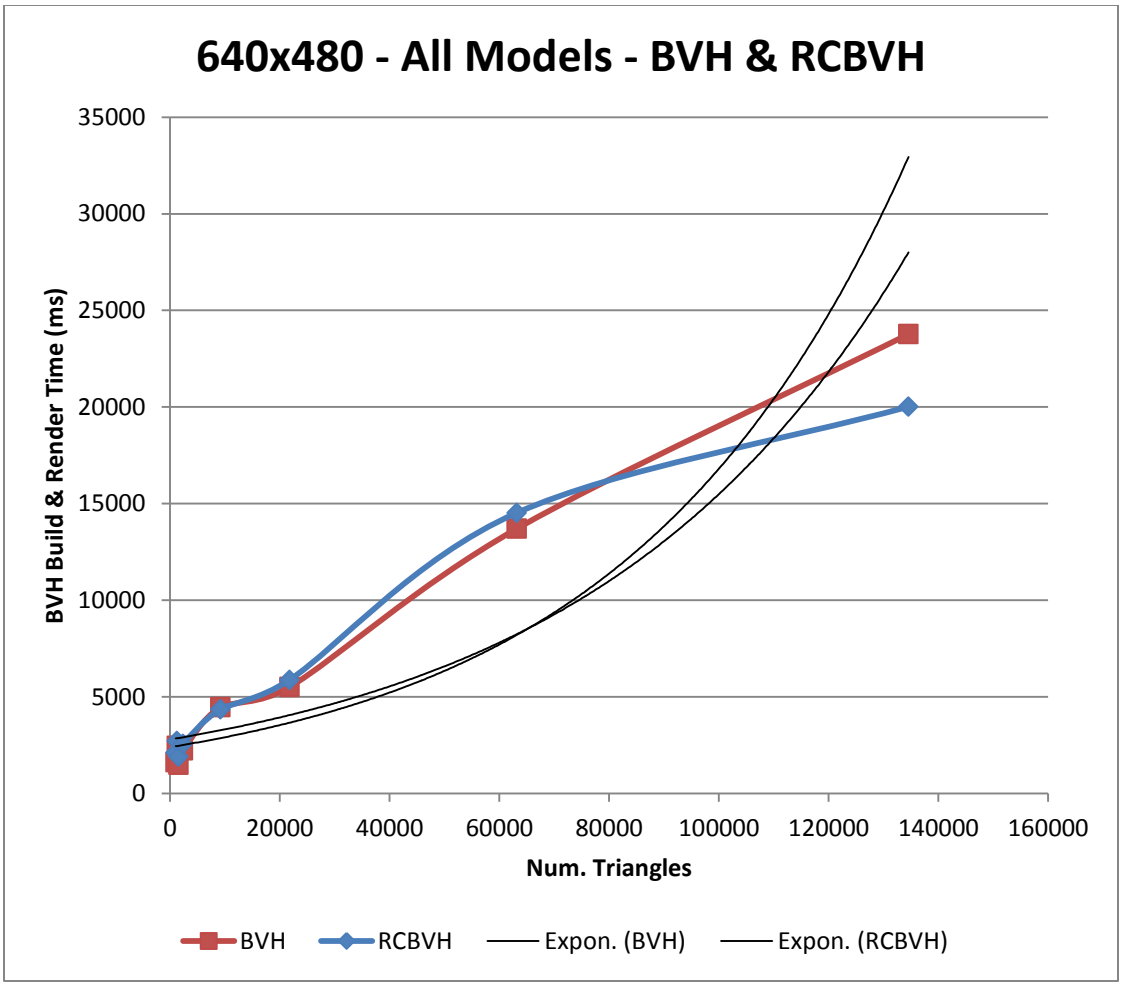

Figure 5: BVH & RCBVH – All Models

The above results show multiple findings. First, as the number of triangles in the scene grows, the amount of time to ray trace the scene grows at an exponential rate. Additionally, the RCBVH algorithm outperforms the traditional means quite handily. The BVH implementation diverges upward at a faster rate than the RCBVH implementation, though both still show exponential trends. The results for the smaller set of triangles require additional exploration as well.

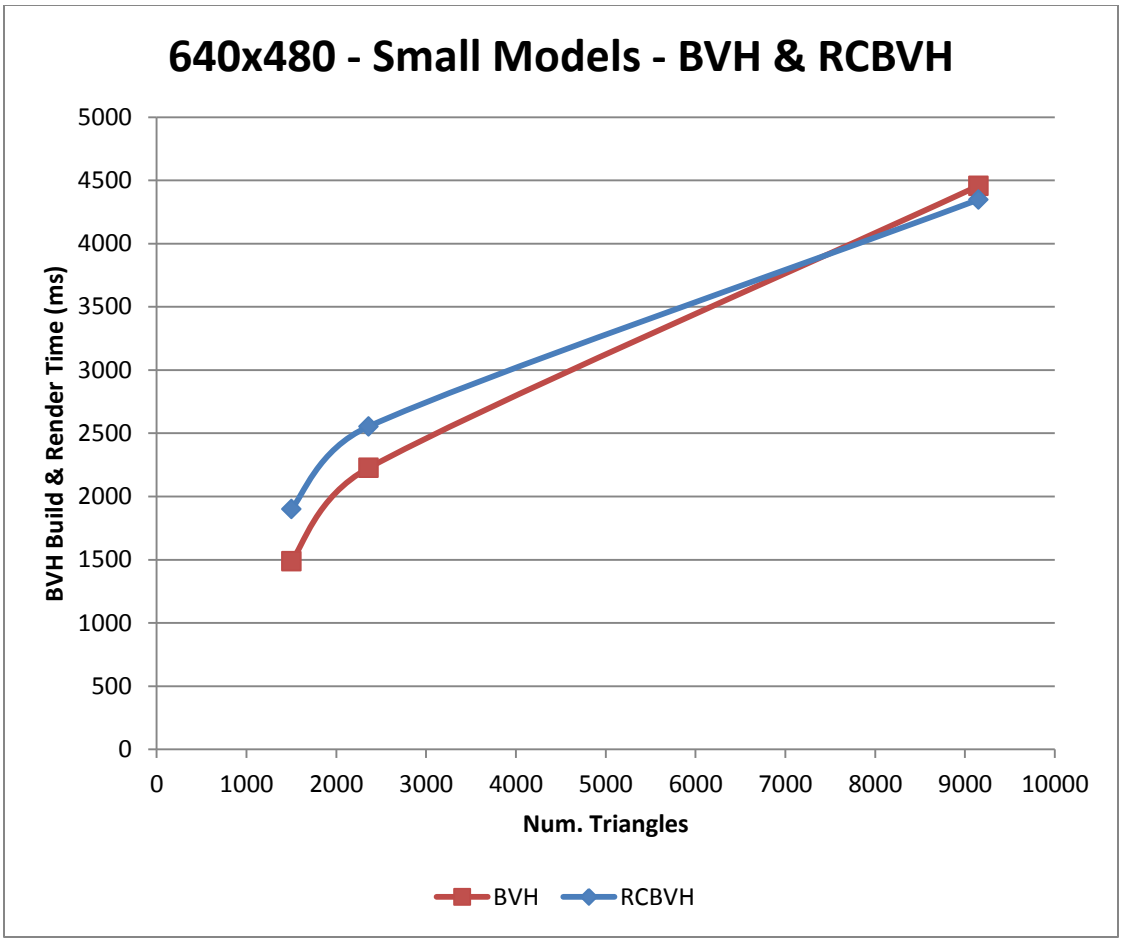

Figure 6: BVH & RCBVH – Small Models

The results confirm something that the author suspected. When the number of triangles being rendered is small, the cost of performing the RCBVH algorithm is greater than creating the entire BVH tree structure. Those costs, though not significant, still give the traditional method an edge over this algorithm. But as the number of triangles in the model becomes greater, the cost of creating the entire tree catches up to the traditional algorithm and the RCBVH begins to excel.

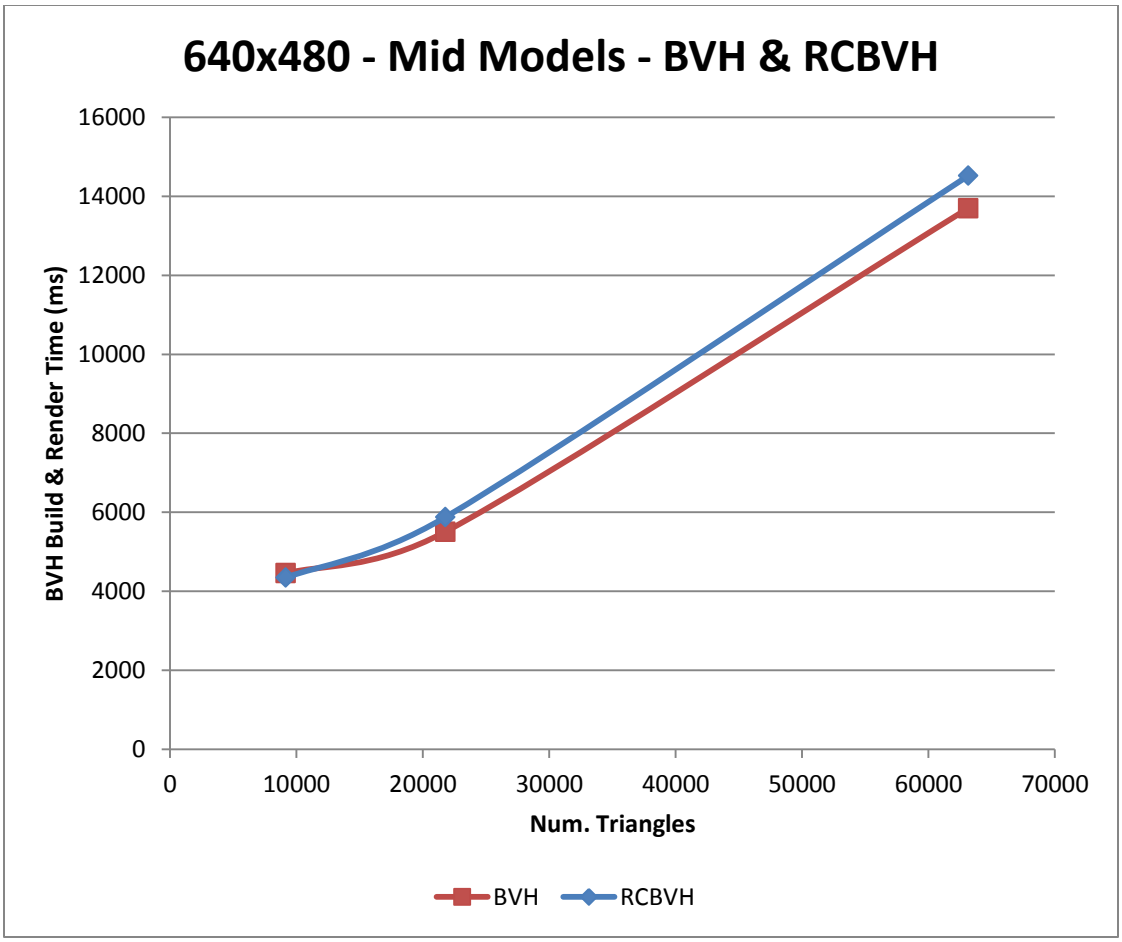

Figure 7: BVH & RCBVH Mid Models

It is easy to see that, from the above results, the RCBVH provides improvements over the traditional way, building only the parts of the BVH tree that are necessary for rendering the scene. This becomes even more apparent as the number of triangles grows to even greater sizes.

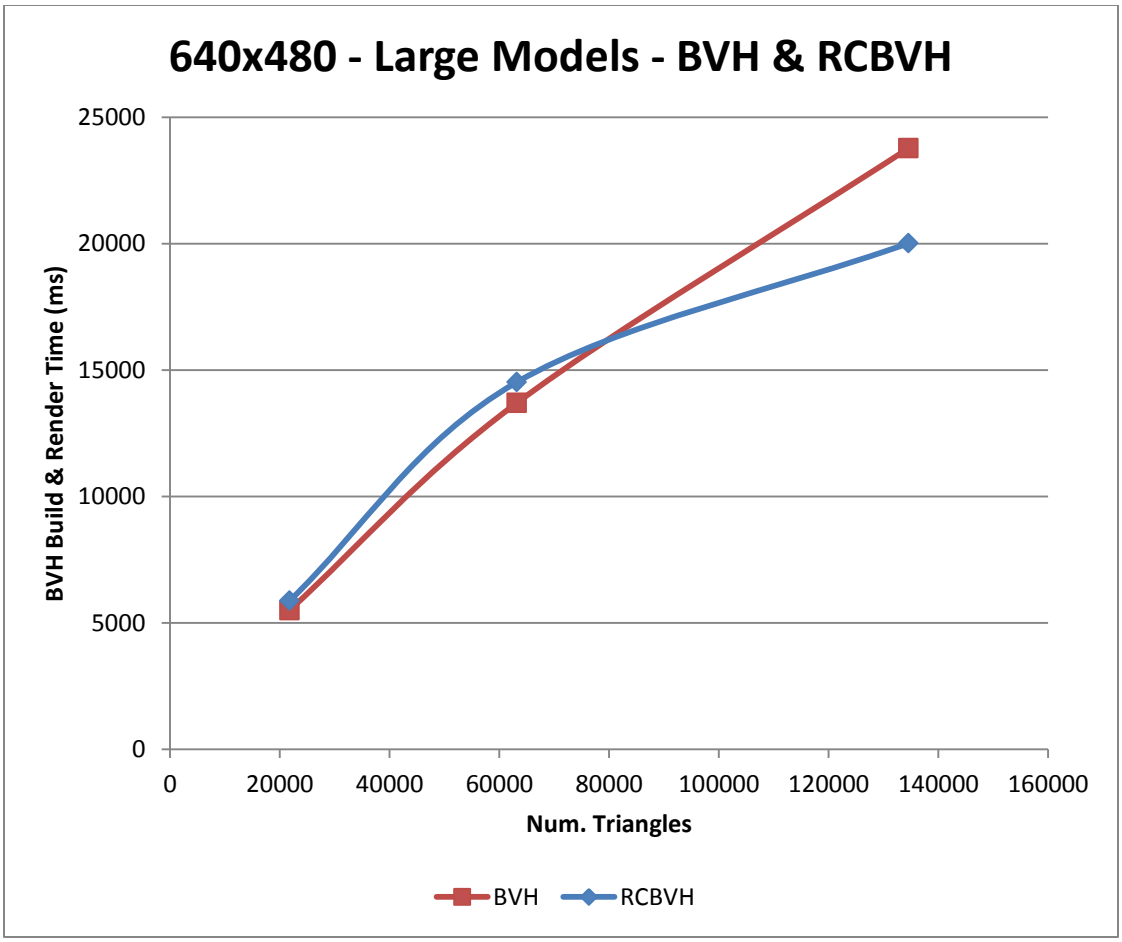

Figure 8: BVH & RCBVH - Large Models

It appears, as the author suspected, that when the triangle count is low, the additional overhead of performing intersections while constructing the RCBVH out-weighs its benefits. As the scene fills with more rays which intersect the model, this drives the cost of building the RCBVH higher, forcing the total time to build the acceleration structure and render the rays to increase.

Table 1: BVH & RCBVH Results

Num Triangles	RCBVH - Time(ms)	BVH - Time(ms)
1010	2112	1617
1246	2705	2449
1500	1899	1487
2360	2554	2226
9150	4348	4457
21772	5877	5508
63156	14519	13697
134576	20015	23770

The following images are rendered in order from simple to complex models.

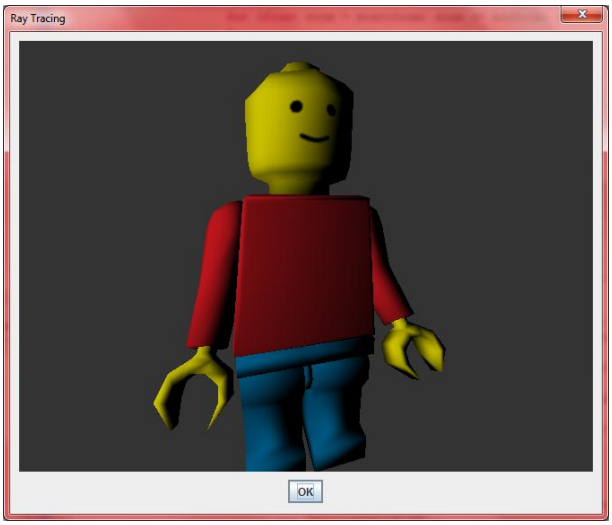

Figure 9: Simple Model Render - ~1K Triangles

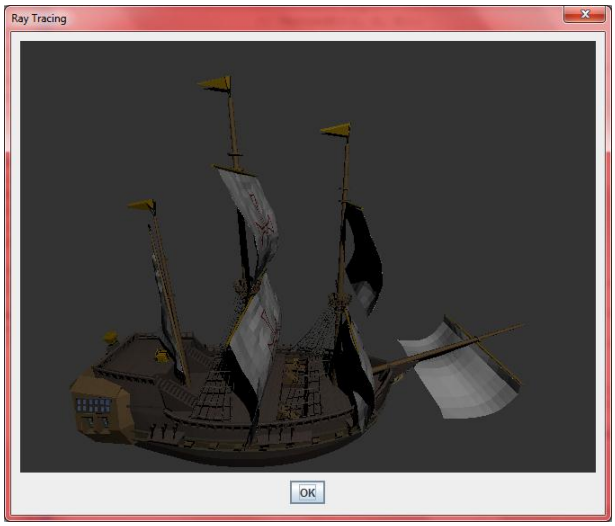

Figure 10: Mid-Sized Model - ~35K Triangles

The RCBVH took 2575ms to render the LegoMan model whereas the traditional BVH took 2137ms because of the lack of triangles and the simplicity of the model.

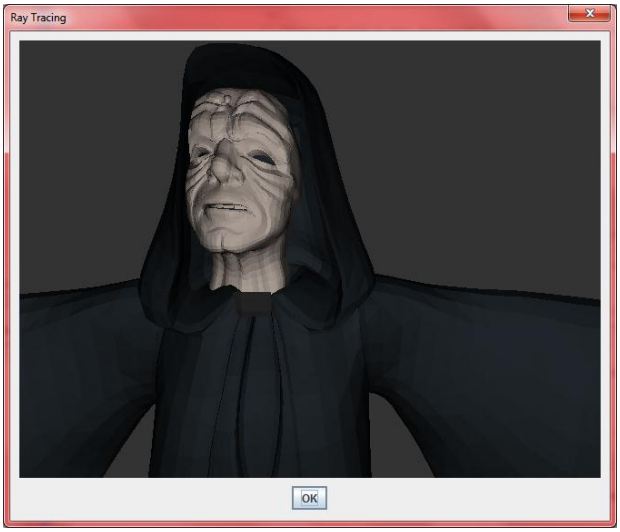

Figure 11: Semi-Complex Model - ~21K Triangles

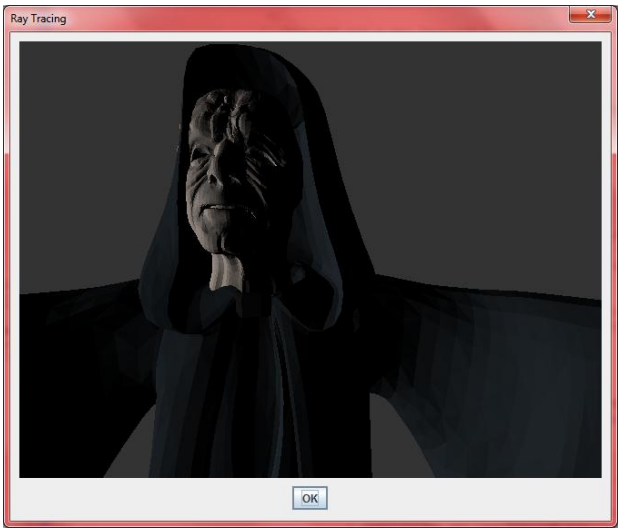

Figure 12: Mid-Sized Model – Shadowing

Discussion

The advantages to this algorithm are twofold. The first advantage being that only the parts of the tree which are required are generated. For a normal scenario, the entire BVH tree needs to be calculated, therefore, if there are triangles which are not intersected by rays, an excessive amount of possibly unused paths will be created. The second advantage is that time traversing similar paths of the tree do not have to be performed. When the BVH tree gets generated fully, that counts as a single traversal. Subsequently, each ray must then traverse the tree and many of the same paths that previous rays had done. Indeed it is unavoidable that all of the intersection tests have to be performed the same amount of times, but at least it is only done once while passing through that segment of the tree, exploiting memory locality of the BVH node for the rays intersecting it. With RCBVH, that only has to be performed once and, exploiting the efficiency of loops, they are used for a large collection of rays. Unfortunately, this

algorithm does not allow the separation of the build process from the render process, since they are now intermixed, but time has been saved not building unnecessary paths and performing many traversals. Additionally, when thinking about memory care/costs, performing this process keeps more memory on the stack while traversing the tree, since the collection of rays is held in memory. Given the testing platform, there is no issue when attempting to record results from the smaller, mid, and even large models. Although, anything past those pushed the limitations of the author's development computer, and valid results were not able to be recorded, but by looking at the trends set from mid-sized to large models, the algorithm would hold quite well, especially since it does not require the full tree to be built to render the scene.

This algorithm makes modifications to current day means of accelerating ray tracing to increase the speed and efficiency as to not waste CPU cycles performing similar tasks multiple times. Additionally, this algorithm is well suited for very large scenes where, if the camera is in a location that is not able to see all of the triangles, the paths to those triangles will never be built. Additionally, depending on the resolution and how detailed the triangles are, if all of the triangles in the scene are viewable by the camera, this algorithm will still limit the size of the tree, since all of the rays in the scene may not intersect all of the triangles in the scene, therefore limiting its size and the number of paths built. Fortunately, the results are favorable and, as the triangles in the scene increase, they showed much increased performance over the traditional means of full tree BVH construction

Ray Collection with Thread Division

After having implemented the RCBVH algorithm, the author decided to attempt to exploit the inherent parallelism in ray tracing by making the algorithm concurrent. Each ray can be processed independent of each other since each ray is responsible for coloring a single pixel. Java provides a very comprehensive concurrent library, which was employed to implement this algorithm. At this point, the author had only limited knowledge of concurrent programming and felt that, given n threads, an n-times speedup could be achieved. To start, the author felt that it would be prudent to simply divide the screen into different sections such that each thread would have a smaller portion of the work load. Since the hardware with which the author was working had 4 cores with hyper-threading for each, eight separate threads were able to be executed at a maximum.

Implementation

The implementation of this algorithm is slightly different than the single-threaded approach. In Java, to spawn new threads, one must build a runnable class which is used in the constructor of a new thread. So, the main portion of the code had to be abstracted into this method. The screen is divided numerically, such that when dividing the total number of rays in the scene (a screen resolution of 800x600 would provide 480,000 rays), and divided by the number of threads that are being used to run the algorithm, which is how many are assigned to a single thread. This then allows for all of the threads to use the RCBVH algorithm on their own threads, therefore, it is understandable why the author felt that the speedup would scale directly to the number

of independent threads of execution available. Unfortunately, this was not correct. Though the rays are completely independent of each other, the BVH tree is a shared resource amongst the threads. Therefore, once run for the first time, the author did not receive any speedup what-so-ever. Though troublesome, it was easily understood what was happening. Each thread went to the blank BVH structure and the first thread to enter the parent bounding volume has to create the children for it, since none exist. But the other already started threads want to do the same thing as the first. Therefore, a lock needed to be placed on the method which creates the children. So the first thread enters and locks the parent from the other threads, and creates the children. Once the children are created, then the other threads are allowed to pass through and check whether the parent volume has children. Another point to note is that, since there is no way around having the remaining threads wait on the first thread to build the children since no additional work is able to be done until the result of that thread is returned.

The results of the execution showed slightly better times, but barely any faster. There was still something that was not functioning properly, but after some careful thought, the author discovered the final piece of the puzzle to this algorithm. The problem occurs that all of the threads get locked on a single volume when waiting for the children to be created. And, if all of the threads follow the exact same path, even after the children are created, they still get locked following the same path. Some threads might exit a part of the recursion early within the process of building the tree and intersecting their respective rays. This is due to the fact that there are no more rays that intersect the bounding volume along that given path, therefore they are free to go

and create other parts of the tree while the others are occupied creating a different branch. This is what was needed. For the final approach to this implementation, a form of a toggle was placed on a BVH node. Once a thread enters, the switch is flipped to signal to the next thread, to start going down the opposite path. Using Java's implementation of AtomicBoolean, it is guaranteed that each thread will atomically get the current path it should take and set it to the opposite of itself. This acts as the switch between left traversal and right traversal. To ensure that the thread paths are "toggled" appropriately, additional care has to be taken at this step. A compareAndSet has to be performed to ensure that, when storing the AtomicBoolean's value and setting it opposite to itself, that the value that was retrieved is still the same value that AtomicBoolean has; then it can be set to its opposite. The first thread will recursively traverse the left child first, then the right, whereas the second thread would traverse the right side, then left. And since this is a per node toggle, this allows for as many threads to go on almost completely unique paths separately. This does not guarantee that the threads will not collide and therefore lock each other from proceeding, especially at the highest levels of the tree, but this allows them to go and build the tree as asynchronously as possible, while at the same time performing the necessary intersection tests along the way. The following pseudo code shows how this implementation functions (For the actual implementation of the algorithm, the reader is referred to appendix – getRCBVHTrianglesAsynchronous()).

```
function recursiveTreeBuild(rays, node)
{
      bool traverseLeft = node.getAndSwitchLeftTravers()
```

```
        ray[] intersectedRays
        for (ray in rays)
        {
              if (node.triangles.size == 1)
              {
                    ray.triangles.add(node.triangles)
              }
              else if(ray.intersect(node)
              {
                     intersectedRays.add(ray)
              }

        }

        if (node.triangles.size == 1)
        {
              return
        }

        if (intersectedRays.size > 0)
        {
              // Locks threads from building children
              node.buildChildrenSynchronous()

              if (traverseLeft)
              {
                    // Traverse Left
                    recursiveTreeBuild(intersectedRays, node.leftChild)

                    // Traverse Right
                    recursiveTreeBuild(intersectedRays, node.rightChild)
              }
              else
              {
                    // Traverse Right
                    recursiveTreeBuild(intersectedRays, node.rightChild)

                    // Traverse Left
                    recursiveTreeBuild(intersectedRays, node.leftChild)
              }
        }
}
```

Figure 13: Psuedo Code of RCBVH with Thread Division

Results

The results from this implementation are tested using the same test parameters as the RCBVH implementation above, but, in addition, a range of threads is tested as

well (2-16). The following is the entirety of the tests performed for the Thread Division implementation.

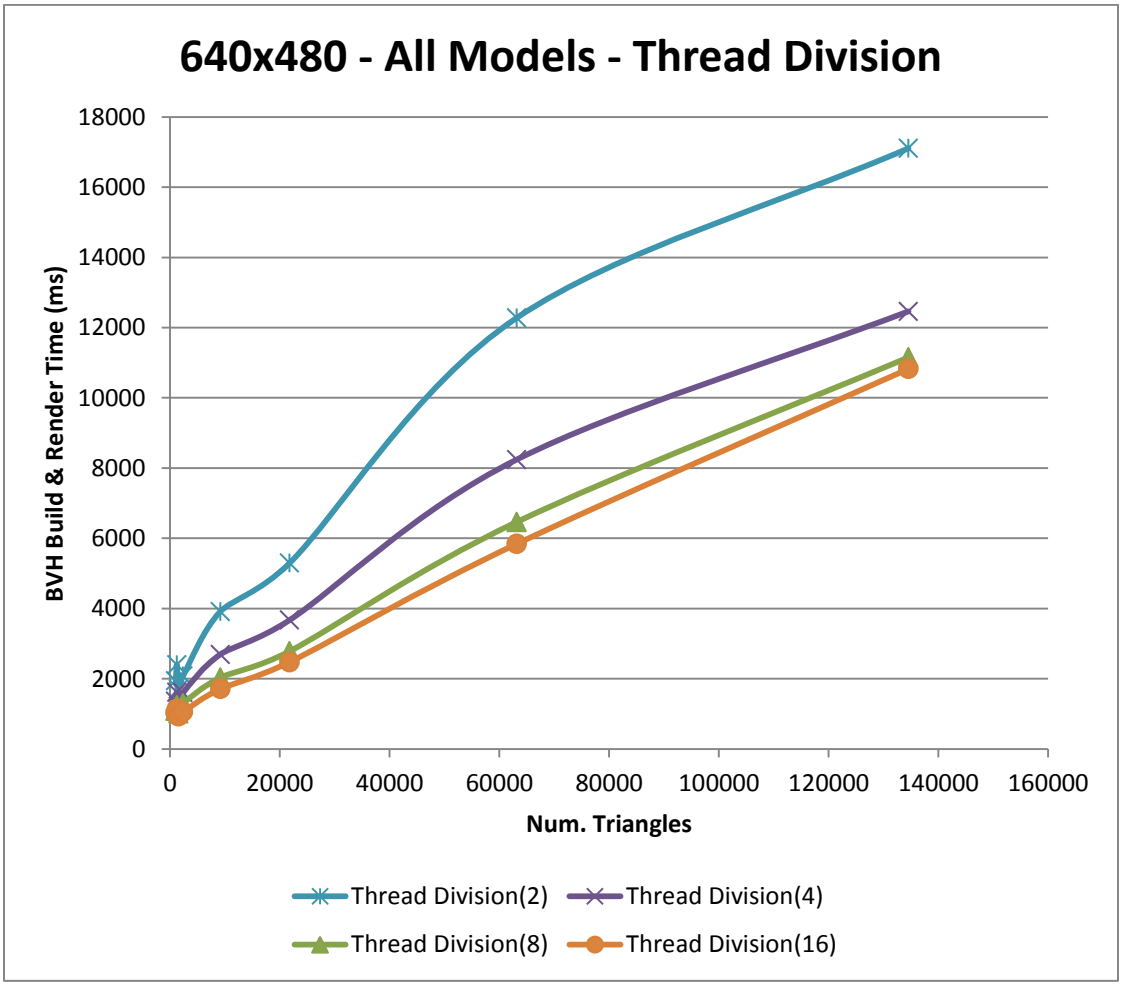

Figure 14: Thread Division - All Models

As seen above, the results of the algorithm prove that by increasing the thread count, the speed of the overall algorithm increases, but it does not increase at the same rate as the amount of threads which are provided to execute the algorithm. One would expect that if the number of threads available to execute concurrently is doubled, the amount of speedup achieved would be doubled as well. Additionally, it can be seen that,

even though there are only 8 threads available to independent execution, that doubling that to 16 still showed improvement over the 8 threads, but not as much as going from 2 to 4 or 4 to 8, respectively. The trends for decreasing the amount of time to render by increasing the number of threads can be seen below.

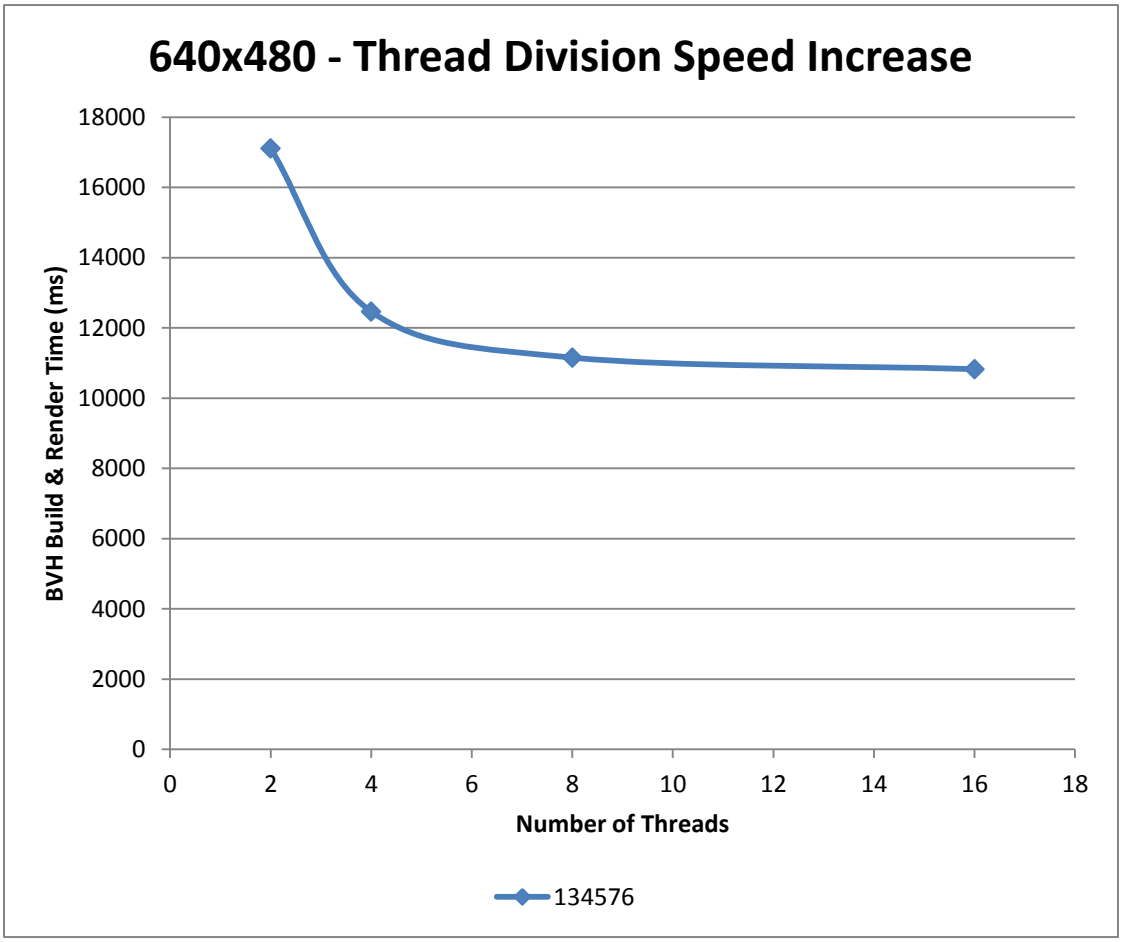

Figure 15: Thread Division – 134K Triangles

Since the model with 279K triangles gives the widest spread of times to render, and is a model which most closely resembles real world models, this was chosen to show the speed increase trends. Moving from 2 to 4 threads doesn't decrease the time by half,

but by a significant amount, but as the threads are doubled, diminishing returns are gained. Again, as the previous algorithm, similar trends are seen with this algorithm, even when the threads are increased.

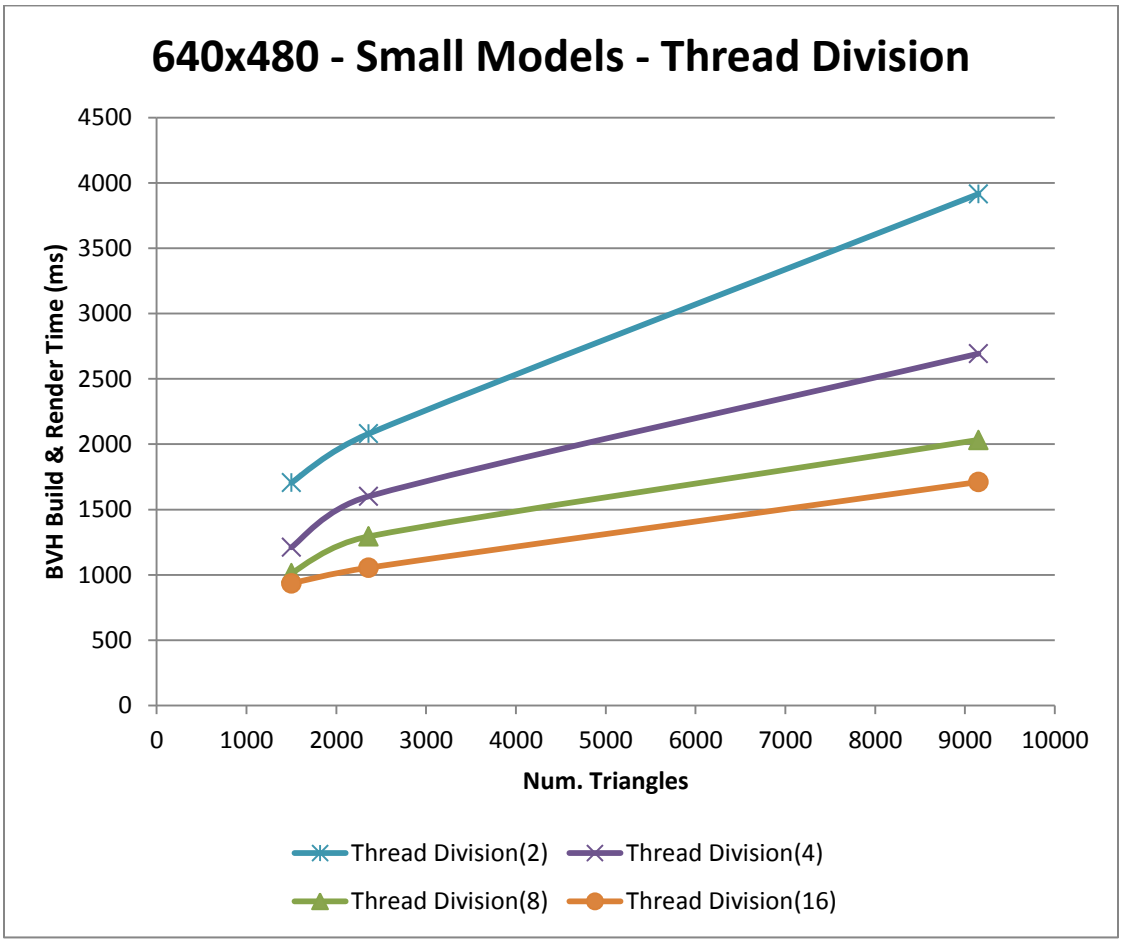

Figure 16: Thread Division – Small Models

As with every implementation to follow, the more threads allotted to the algorithm, the better it performs. All trend the same, but all perform better than the smaller thread count execution.

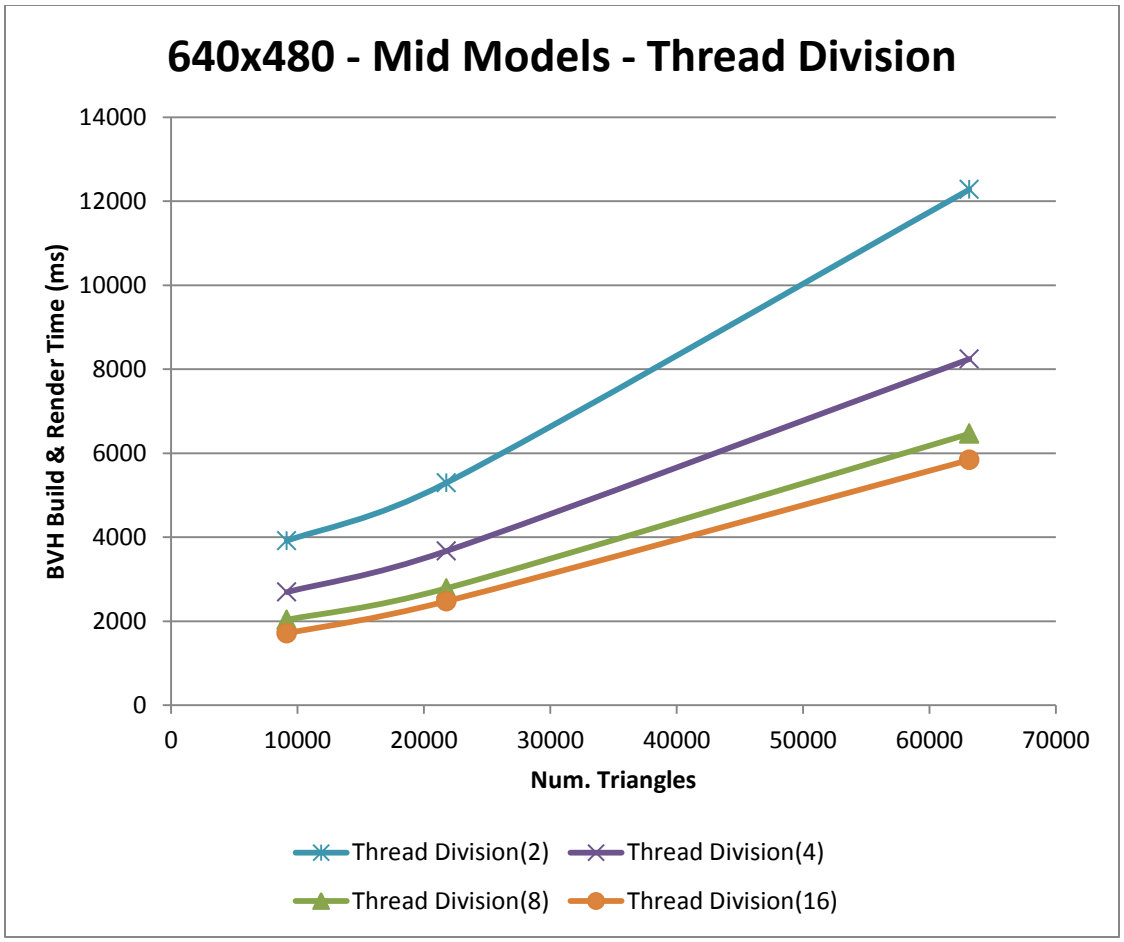

Figure 17: Thread Division – Mid Models

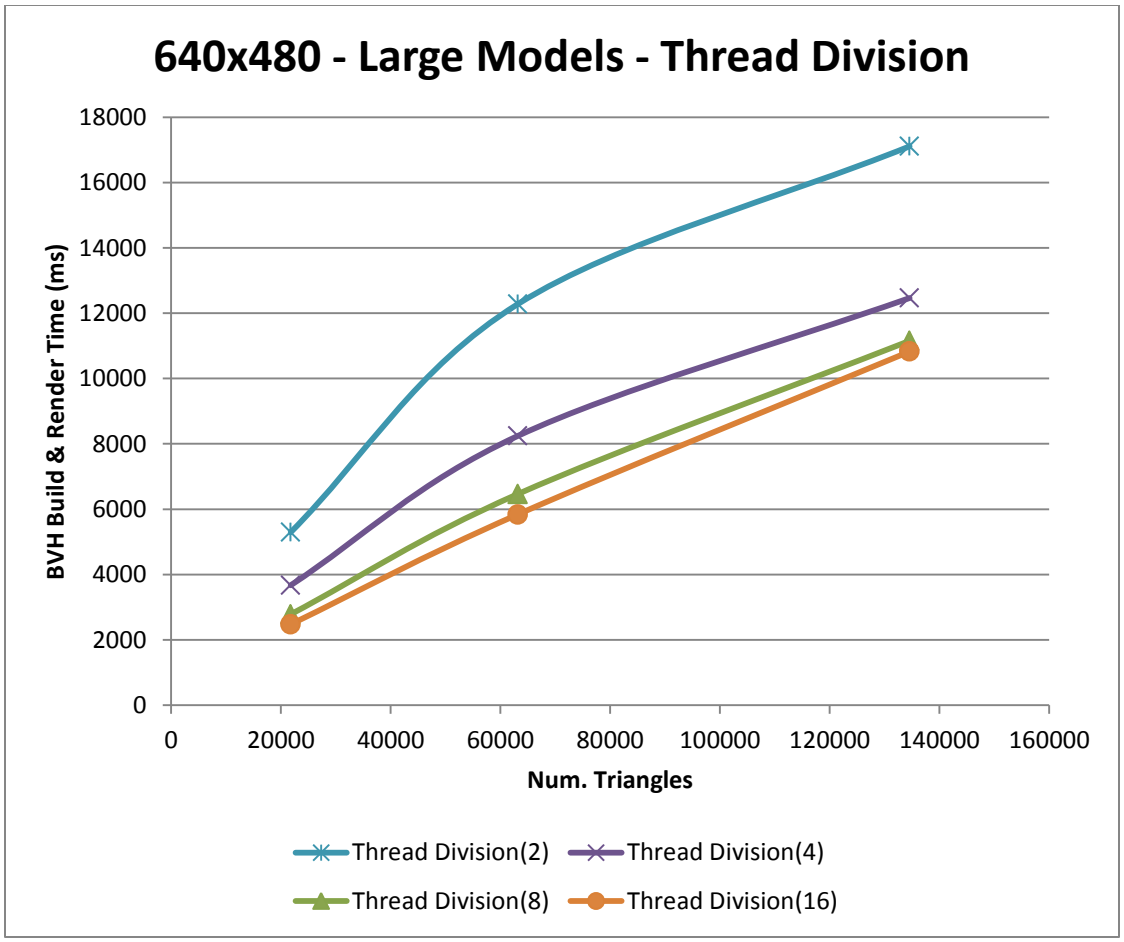

Figure 18: Thread Division – Large Models

The results of this algorithm are, as with the basis RCBVH, following similar trends. The lines flatten out a little more as more threads are introduced, but all the same shape. Below are the timing results for the above graphs.

Table 2: Thread Division Results

Thread Division – Time (ms)				
Num Triangles	2 Threads	4 Threads	8 Threads	16 Threads
1010	1944	1357	1086	1029
1246	2402	1611	1237	1144
1500	1705	1211	1013	935
2360	2080	1601	1294	1055
9150	3915	2693	2033	1711
21772	5293	3671	2782	2475
63156	12277	8242	6463	5839
134576	17108	12464	11154	10826

The following are a few renders of mid and large sized models.

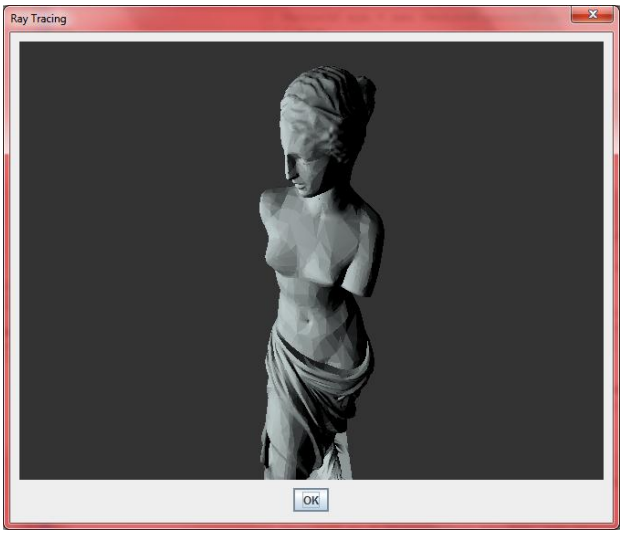

Figure 19: Mid-Sized Model - ~43K Triangles

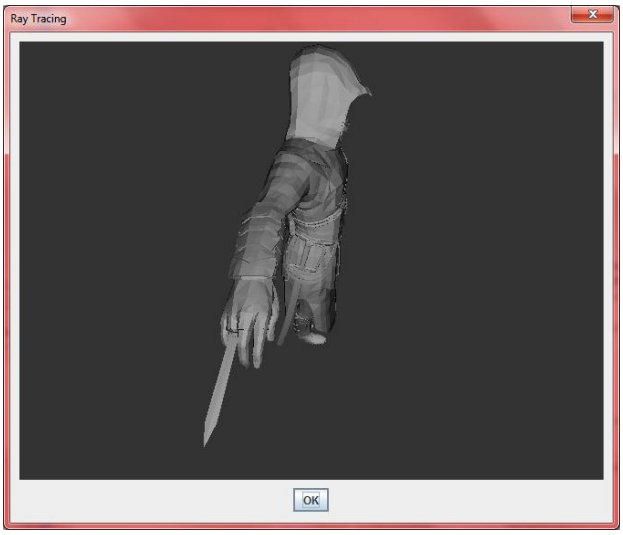

Figure 20: Mid-Sized Model - ~23K Triangles

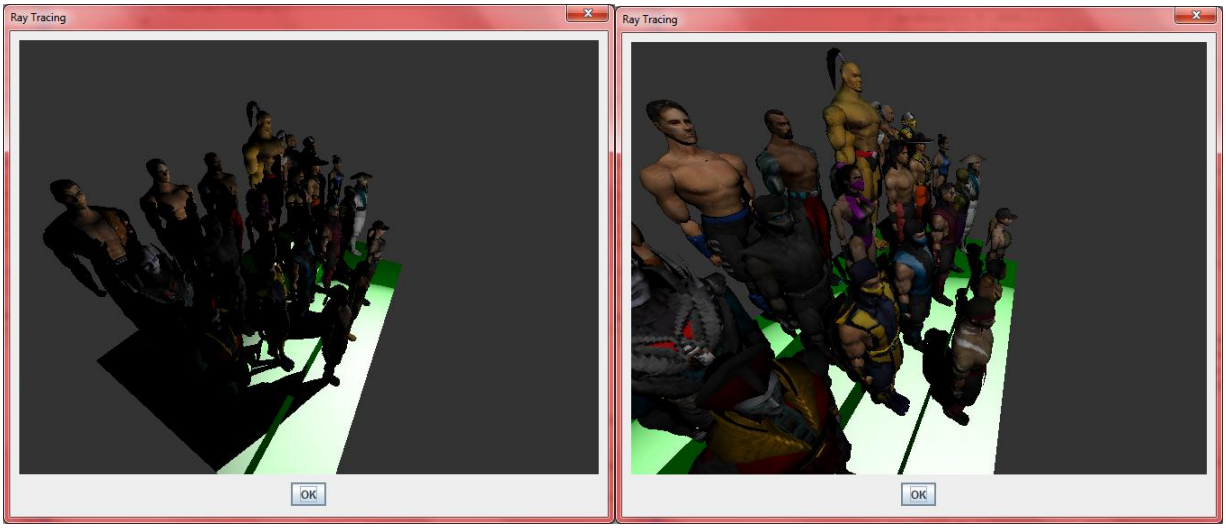

Figure 21: Large Model - ~96K Triangles

Figure 22: Large Model - ~96K Triangles

Comparison

Of particular interest, even more than the trends of how additional threads affects the result, is the comparison of this, supposedly twice or more times better algorithm and the previous single threaded approach. This is not always the case.

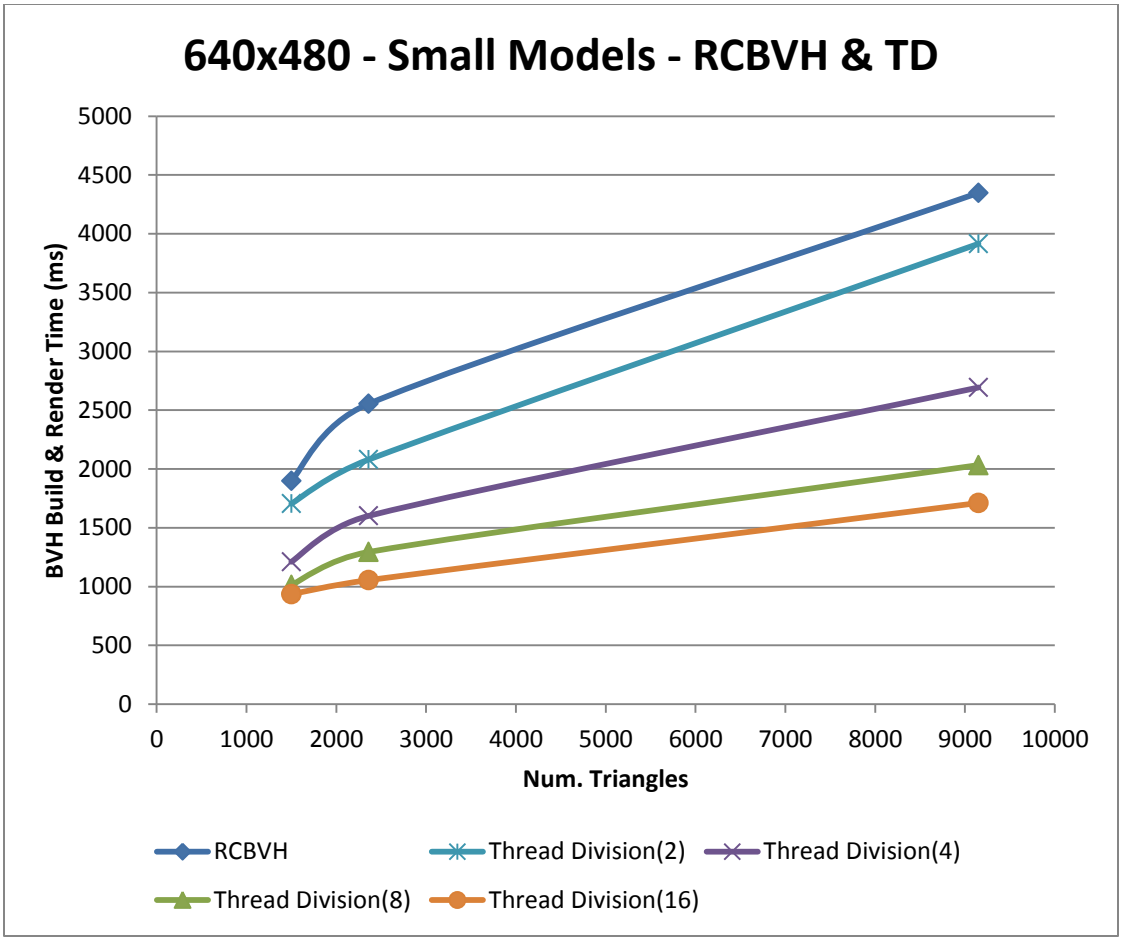

Figure 23: RCBVH & TD - Small Models

Above, the comparison of the small model renders shows that the Thread Division algorithm performs better single threaded RCBVH implementation. And, of course, as the threads increase, the time to perform the same operations decreases, becoming faster.

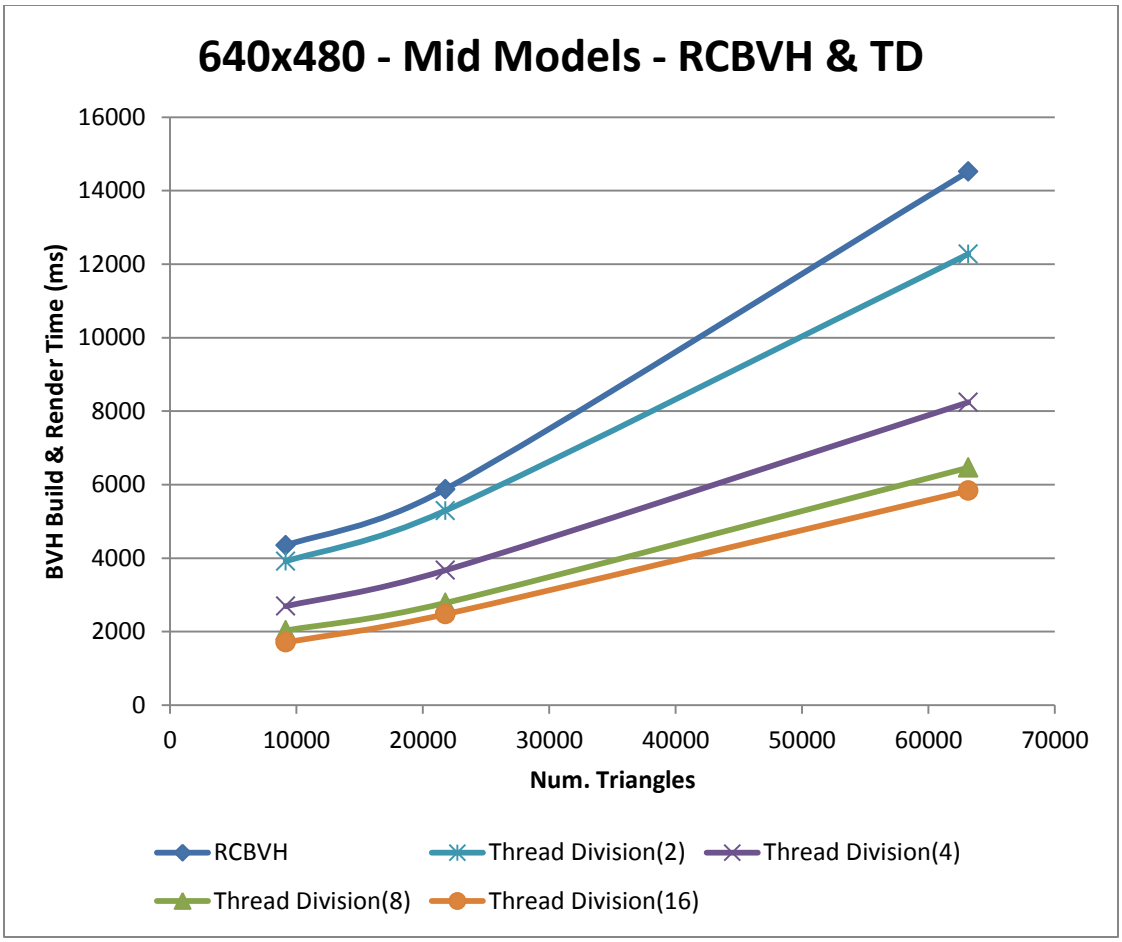

Figure 24: RCBVH & TD - Mid Models

In the mid-sized model renders, the same trends occur. Though faster, the two threaded Thread Division is not twice as fast as the single threaded approach.

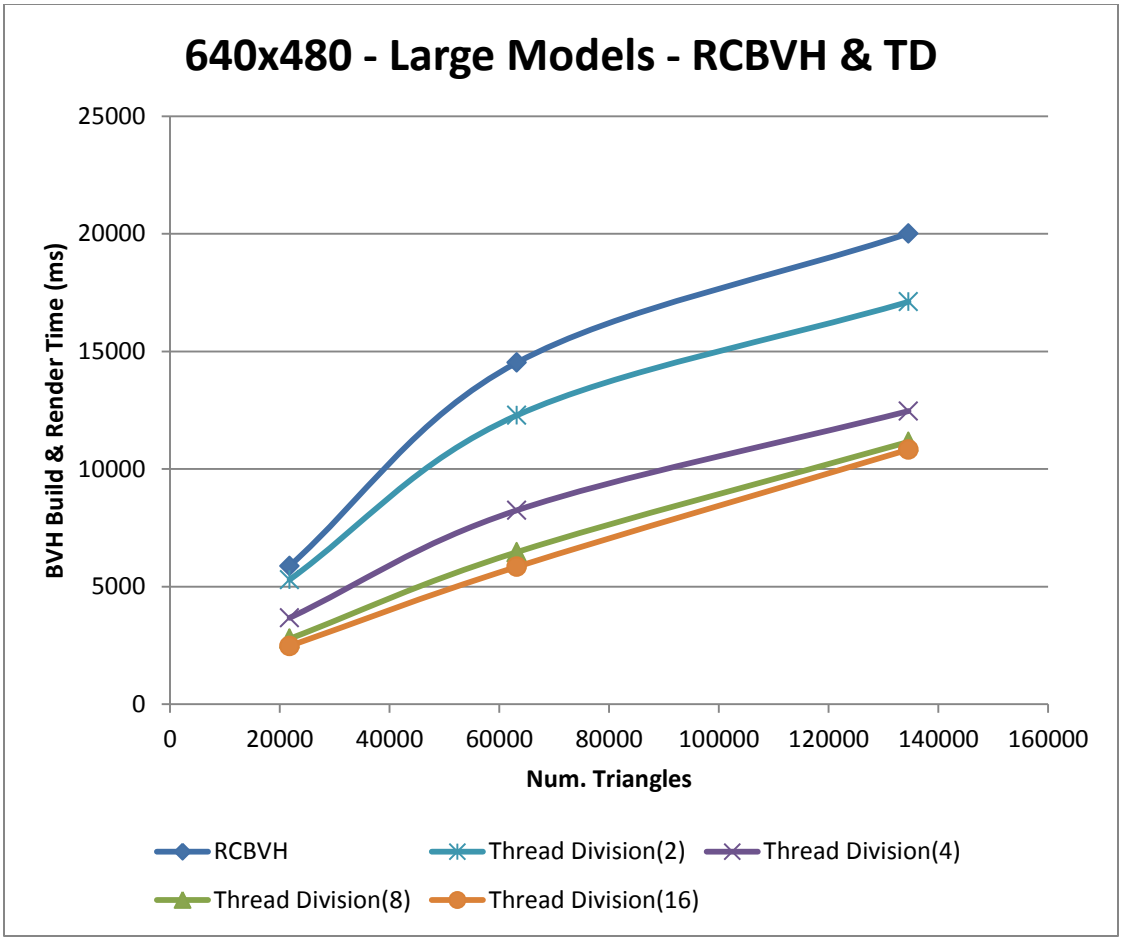

Figure 25: RCBVH & TD - Large Models

When reaching to the largest models rendered, again, the same behavior occurs. As expected, the multi-threaded approach continuously outperforms the single threaded approach.

Discussion

This algorithm proves that, given more threads, the original RCBVH algorithm can be improved. Discouraging, though, is that the speedup was not n times. The advantages to this approach are that not only does the tree get built in parallel, but it is

performs much faster since separate threads are able to execute their builds on separate parts of the tree simultaneously. The disadvantages, besides the tree being a shared resource, is that, regardless of the implementation, all threads have to wait for the first set of children to be built. This is unavoidable, and therefore, no work can be completed until this happens.

Of additional discussion is the lack of n times speedup given n threads of execution. The author feels that this occurs due to a similar case as the Single Threaded RCBV approach runs into additional overhead. For the single threaded solution, the overhead comes in the form of performing the triangle intersections while creating the node children. In this approach, the overhead comes from thread synchronization. To ensure that threads get the necessary values to perform at their best and most efficient, additional synchronization overhead has to be performed which hurt the overall outcome of the algorithm. Even though each thread has a part of the tree to work on, having to pass through synchronization locks, atomic operations, and waiting for the other to build children slows down the algorithm significantly. But as the amount of threads are increased, the results are far more encouraging and, since there are so many threads working on the tree concurrently, the cost of synchronization is absorbed quite a bit. Again, this will not be the norm and generally very large scenes are used and this algorithm would flourish quite well over the traditional approach and definitely over the single threaded version as well.

Ray Collection with Thread Division and N Children BVH

In this algorithm, the author felt that there was definite room for improvement over the previous algorithm. The author had to find a way to identify the flaws in the previous algorithm to improve on them. Since the actual ray tracing portion of the algorithm needed not be modified, since that portion is completely parallel, the portion which needing tuning was access to the BVH structure. The author therefore attempted to implement the same BVH structure, but instead of splitting the parent BVH node into two children, it would be split into n children, as to allow all of the threads to immediately split from the parent and operate independently of each other.

Implementation

The implementation of this algorithm is not so different from the implementation of the previous algorithm. The modification comes, not so much to the RCBVH portion, but to the implementation of the BvhNode object. Instead of having just a left and right child reference, there is a Collection of children. Therefore, depending on the point of execution, either n thread children can be created or only two, as to continue to take advantage of the SAH algorithm for splitting the nodes, but also attempting to separate the threads of execution from the beginning. Therefore, the triangles within the parent BVH node are sorted along a specific axis and then split numerically based on the number of triangles and the number of threads being executed. The triangles are sorted along an axis because, if bounding volumes containing the subdivided triangles are overlapping each other, then the advantage of building a bounding volume hierarchy is partially nullified, since rays attempting a ray-AABB intersection may unnecessarily

intersect multiple volumes. After the n children are created, the threads waiting at the synchronization lock pass through and have to decide on which of the paths to take first. On this version of the BvhNode object, there is a Java AtomicInteger. It starts at 0 and, as each thread performs a getsAndIncrement on it, the value they get is the index of the children at which they start. It is guaranteed that each thread will get a unique integer due to the atomic operation being performed. A slight issue arises with this approach, though. If there are more threads than there are children at a certain node (which will most likely always be the case), then if a thread increments the index to a size that is larger than the size of the children, an index out of bounds exception will get thrown. It may seem simple just to set it back to 0, but another issue arises from that as well. If multiple threads get to the same BVH node at the same time and each get the index and determine that the index is greater than the number of children and simply chose index 0, then they are all locked with each other and have to wait for other children to be created. Therefore, the following has to be done: the value of the index is stored using the getAndIncrement method. Inside a while loop, the stored value is checked to see if it is greater than or equal to the size of the children. If it is, then this thread sets the value of the index to 0 using the atomic set method. Then, the same process occurs, where the value of index is stored to the same variable using the getAndIncrement method. If, again, this while loop is executed, meaning that the value retrieved from getAndIncrement is still greater than or equal to the size of the children, then any number of other threads, in between setting the index to 0 and retrieving the new value, called getAndIncrement, making the current thread have to perform that while loop

again. And even though it seems like this is just wasted CPU time, the fact that the current thread has to repeat that portion of code means that at least one other thread was able to proceed. This, therefore, guarantees that each thread will get distributed properly in a round-robin fashion. After the correct value for starting index is retrieved, an additional while loop is used to increment the local copy of this integer until it reaches the end of the children and, if necessary, loops back to the beginning of list of children. Therefore, to be able to index into the list of children, order must be maintained and either a queue or list implementation must be used. After the first level of children, the algorithm switches back to creating others using the same process as before of SAH. Please refer to the pseudo code below for this implementation (and for the full implementation, the author refers the reader to the getRCBVHTrianglesAsynchronousNChildren code in the appendix).

```
function recursiveTreeBuild(rays, node)
{
	ray[] intersectedRays
	for (ray in rays)
	{
		if (node.triangles.size == 1)
		{
			ray.triangles.add(node.triangles)
		}
		else if(ray.intersect(node)
		{
			intersectedRays.add(ray)
		}

	}

	if (node.triangles.size == 1)
	{
		return
	}

	if (intersectedRays.size > 0)
	{
```

```
                // Locks threads from building children
                if (node.buildNThreadChildren.get)
                {
                      node.sortAndSpliPrimitivesSynchronous(n)
                      node. buildNThreadChildren.set(false)
                }
                else
                {
                      node.buildChildrenSynchronous()
                }

                int size = node.children.size
                int index = node.startIndex.getAndIncrement

                while (index >= size)
                {
                      node.startIndex.set(0)
                      index=node.startIndex.getAndIncrement
                }

                int childrenTraversed = 0;

                while (childrenTraversed < size)
                {
                      // Traverse
                      recursiveTreeBuild(intersectedRays,
                      node.children[index])

                      childrenTraversed++
                      index++
                      index %= (size-1)

                }
          }
    }
```

Results

The results for this algorithm were run on the same testing parameters as the Thread Division test. They were simply run for the same models and number of threads for direct comparison.

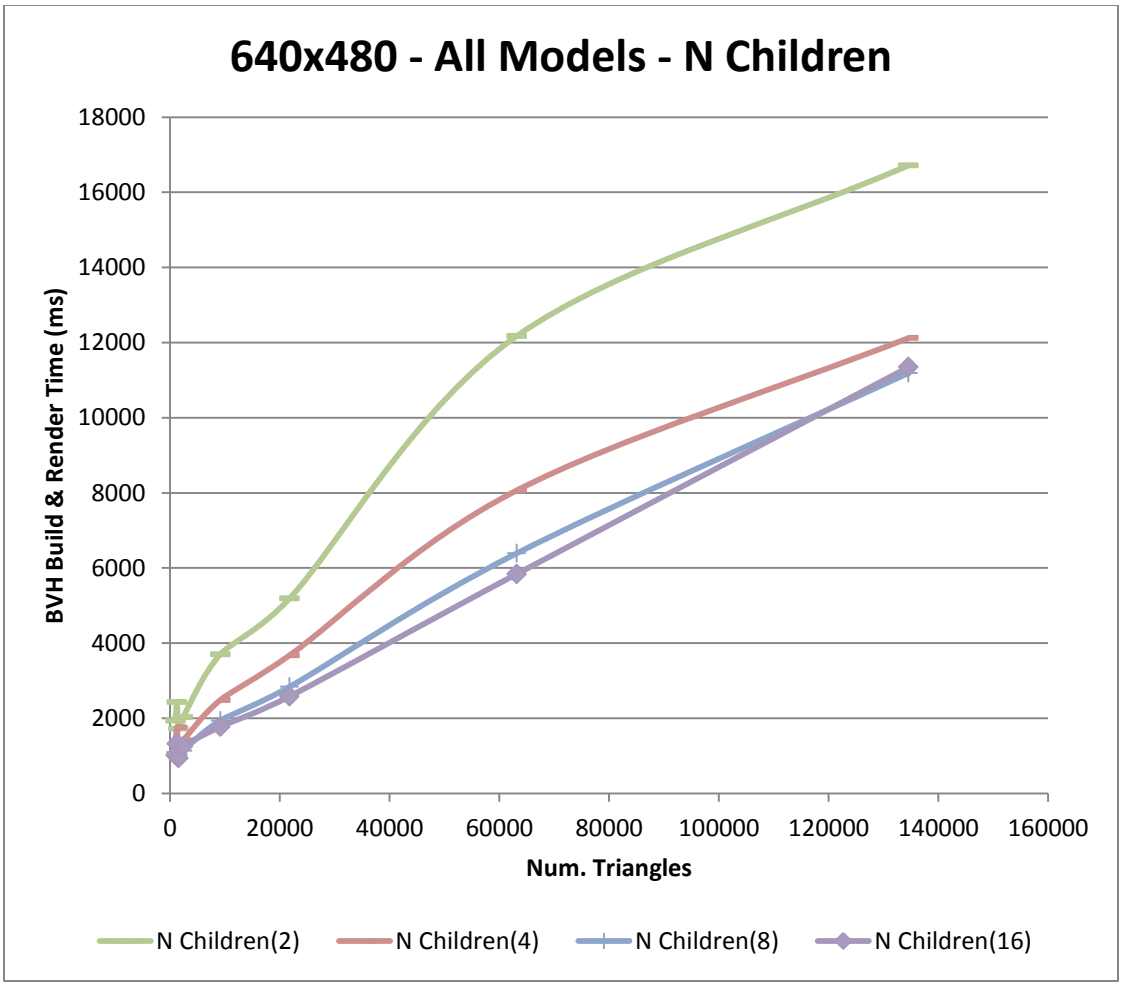

Figure 26: N Children - All Models

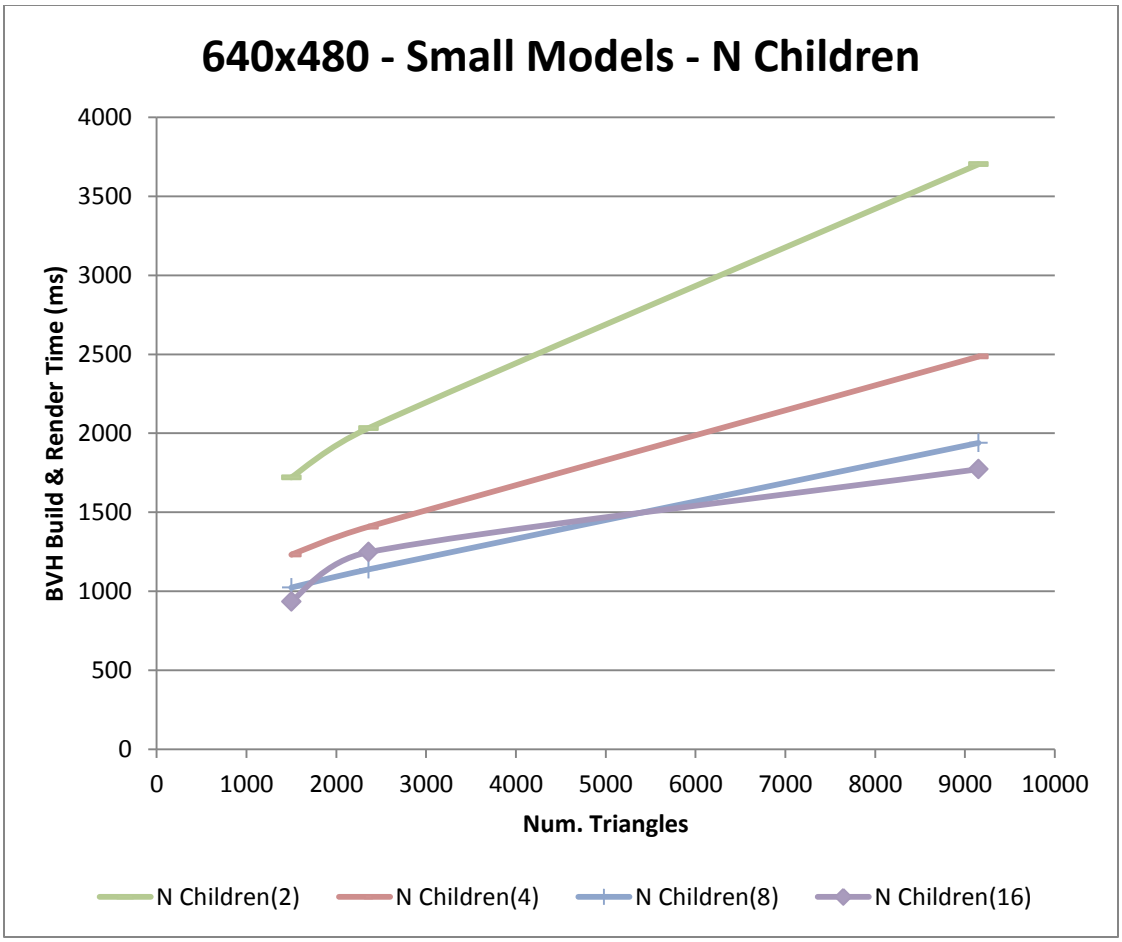

Figure 27: N Children - Small Models

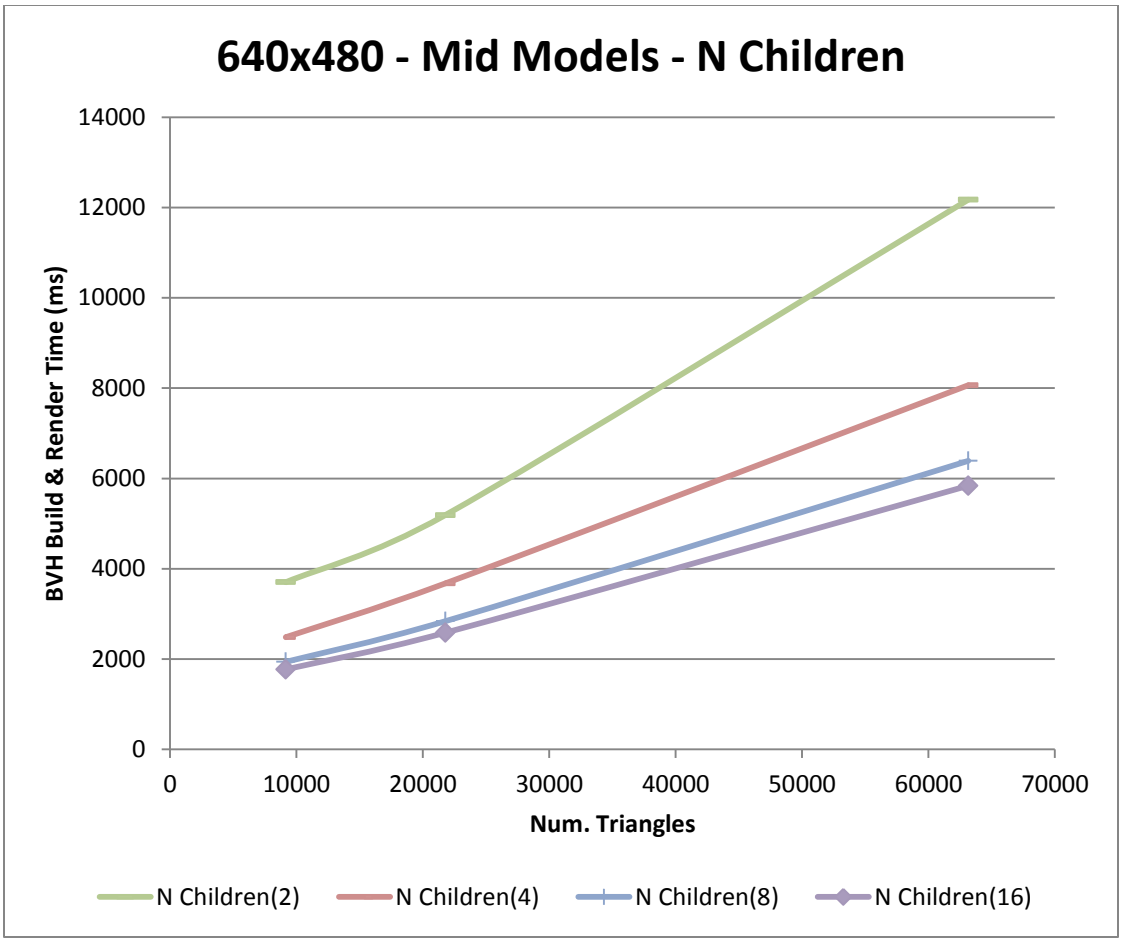

Figure 28: N Children - Mid Models

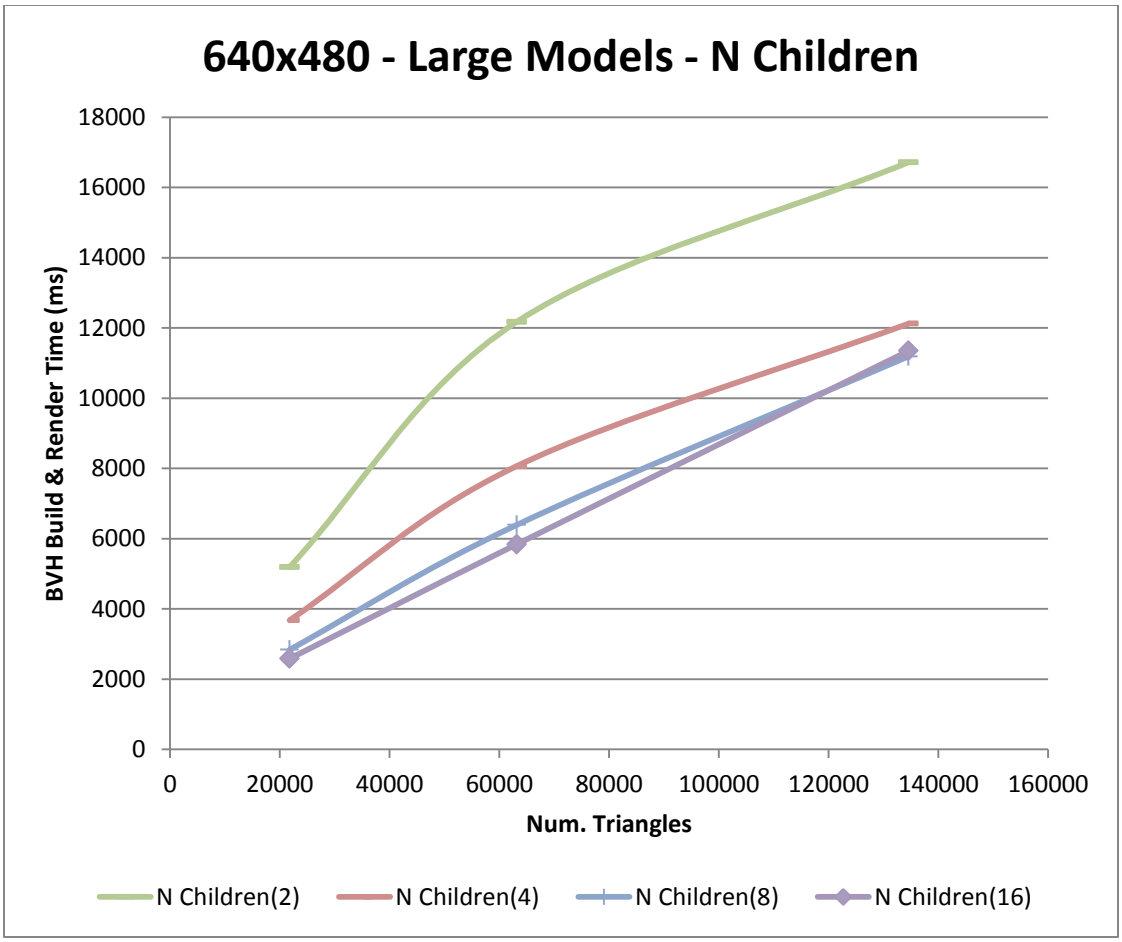

Figure 29: N Children - Large Models

By themselves, the results paint almost the exact same picture as the Thread Division results. As the number of threads double, the results do improve, but not at the expected twice amount.

Table 3: N Children Results

N Children – Time (ms)				
Num Triangles	2 Threads	4 Threads	8 Threads	16 Threads
1010	1940	1367	1097	1014
1246	2428	1757	1305	1321
1500	1721	1232	1024	936
2360	2033	1409	1139	1248
9150	3702	2485	1939	1773
21772	5189	3676	2839	2584
63156	12173	8065	6390	5839
134576	16717	12121	11190	11346

The following are screen captures of some renders from mid to very advanced models.

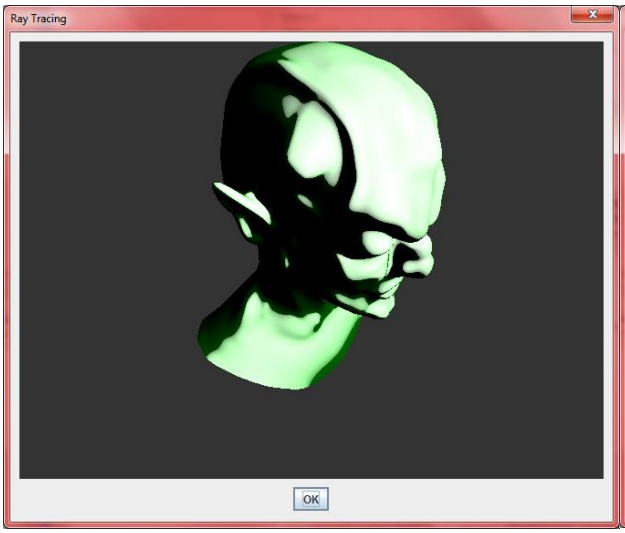

Figure 30: Large Model - ~64K Triangles

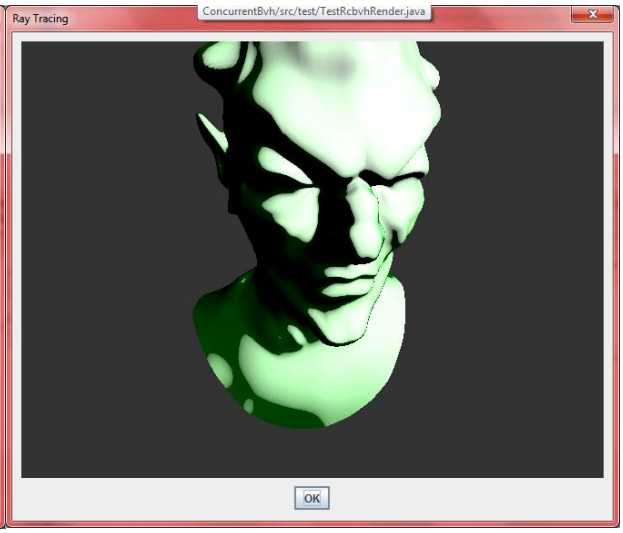

Figure 31: Large Model - ~64K Triangles

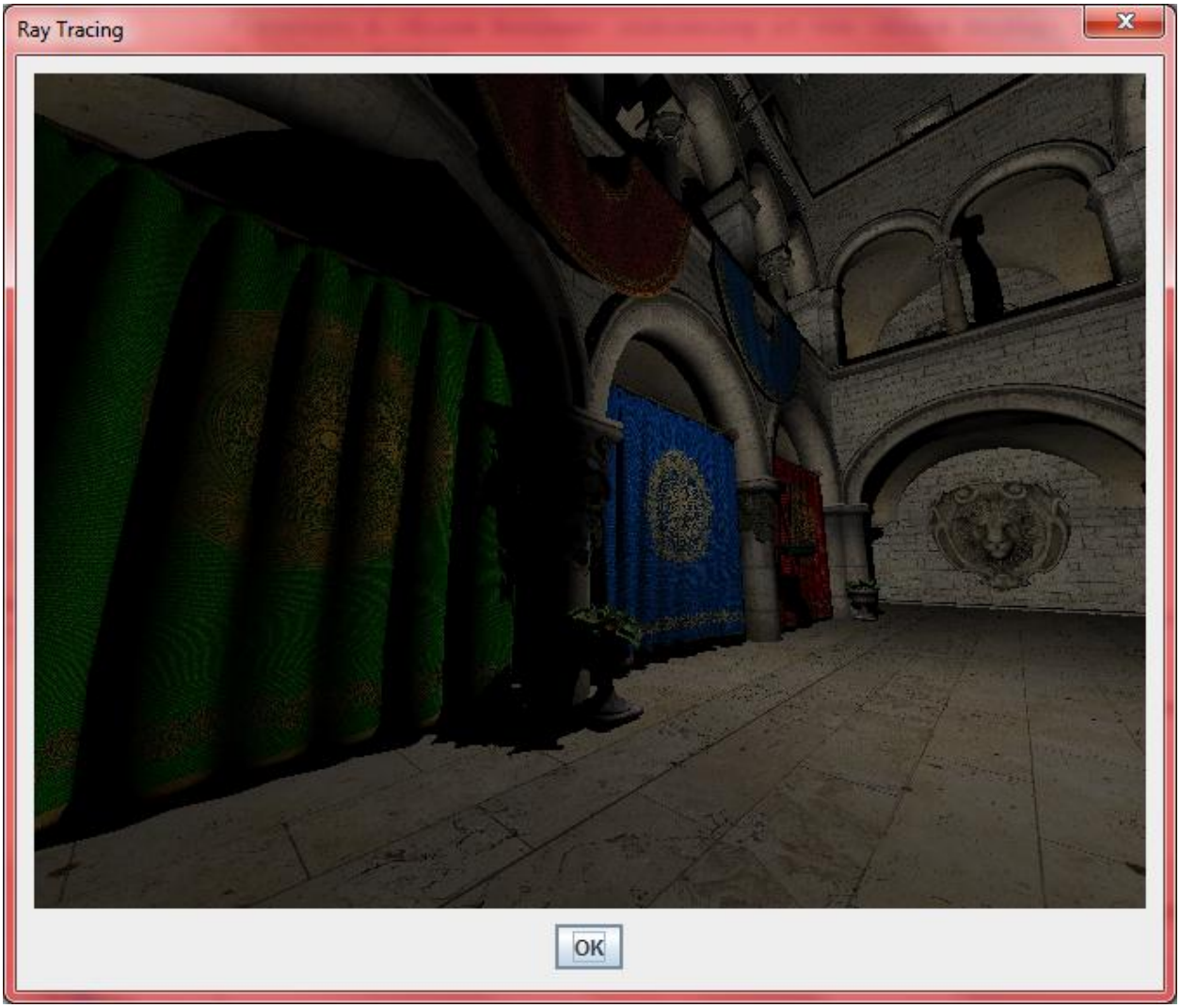

Figure 32: Complex Model - ~279K Triangles

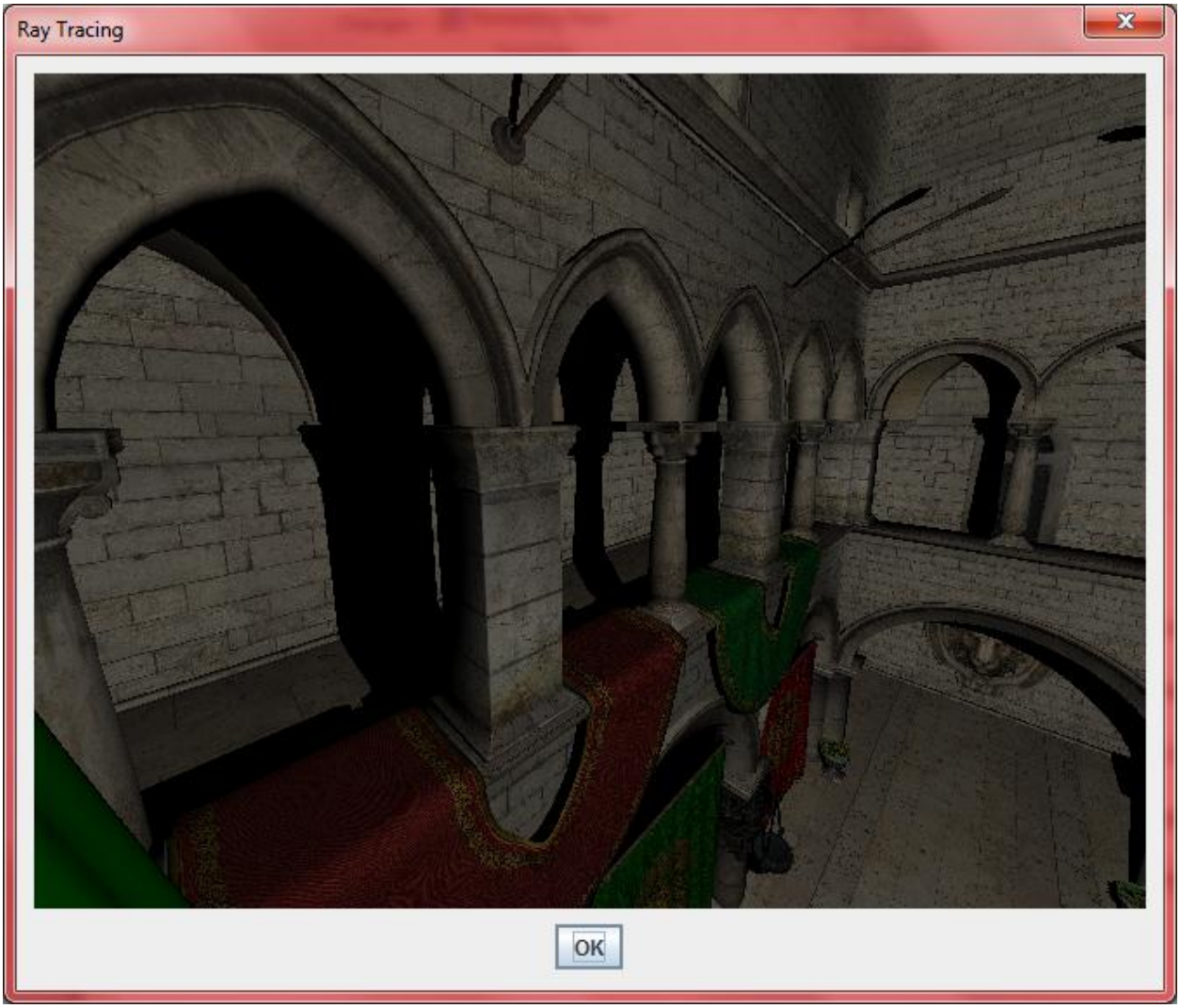

Figure 33: Complex Model - ~279K Triangles

Though these complex scenes with 279,000 triangles were able to be rendered, the results of these scene renders were not recorded due to the memory limitation of the author's laptop. Once the memory limit is reached, the Java Garbage Collector does not sufficient memory to work appropriately and therefore forces excessively longer render times to occur, but the number of BVH nodes created by the BVH and RCBVH algorithms is of importance. The traditional BVH algorithm created, as expected, 2N-1

nodes, which is 558,349 of them whereas the RCBVH algorithm only created only 232,979 nodes.

Comparison

Since the N Children implementation performs on the same level as Thread Division does, it is more prudent to compare these two concurrent ones instead of comparing it with the single threaded implementation, since that was already done in the Thread Division's comparison section. The following graphs show the results for both small scale and large scale models.

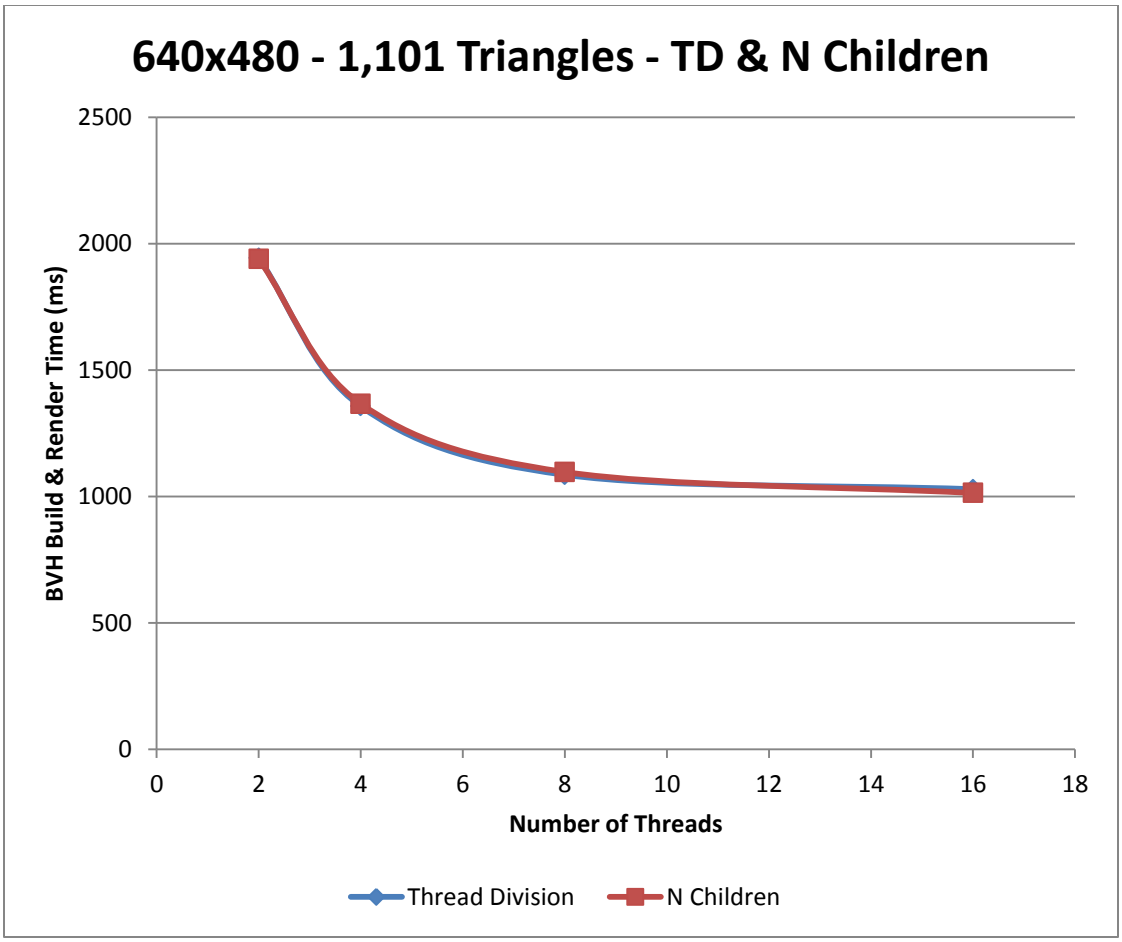

Figure 34: Thread Division & N Children - 1K Triangles

It is simple to see that both algorithms perform almost exactly the same given a very small model to render. The same seems to go as the more triangles are rendered in the scene.

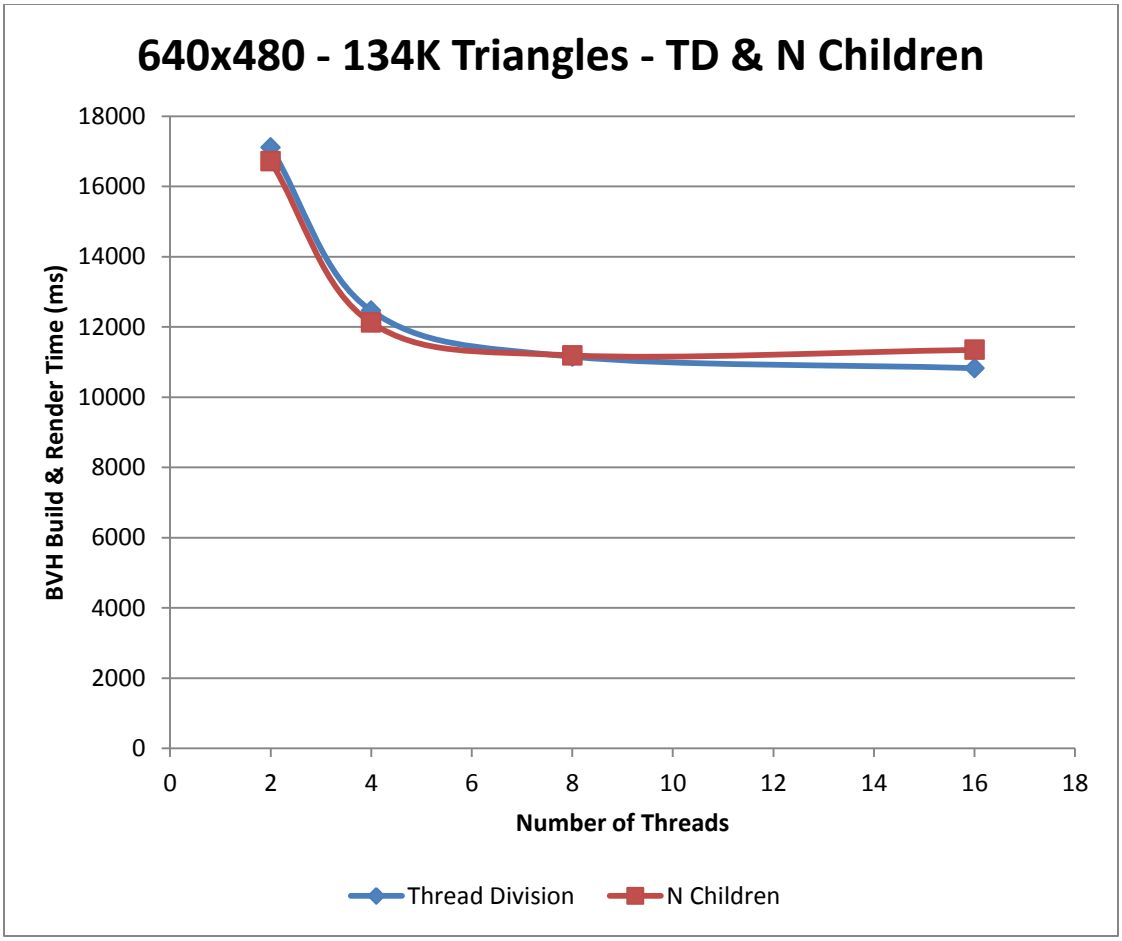

Figure 35: Thread Division & N Children - 134K Triangles

In direct comparison, the N Children method performs ever so slightly better than the Thread Division algorithm. The differences are almost completely negligible, even with a much higher triangle count.

Discussion

To the author, it was puzzling why this method did not outperform its predecessor. This method was supposed to outperform the Thread Division quite handily, but did not for reasons the author could only surmise. Supposedly, allowing all

of the threads to operate independently of each other would be a boon to this algorithm, but the results speak differently. The most probable reason for this slowdown would be due to having to sort the triangles. If the triangles are not sorted, then the distribution of them cannot be guaranteed, and, therefore, lead to far too much overlapping of the BVH nodes. This would then lead to excessively terrible results for rendering the rays through the structure. Unfortunately, this may be the step that slows down the algorithm making this just as fast as the Thread Division one. But the results from this algorithm are still good. There simply might have to something has to be tweaked to allow it to gain an edge over Thread Division.

CHAPTER FOUR: Conclusion

Summary

The Ray Collection Bounding Volume Hierarchy algorithm provides Ray Tracers the ability to only build a sparse acceleration data structure for dynamic scenes to begin ray tracing as fast as possible. Since all of the nodes of the structure do not get built, due to the RCBVH algorithm detecting which branches of the structure need to be built, acceleration on the overall process of can be achieved. When the scene is very simple and does not contain so many triangles, the traditional method of BVH construction and rendering overtakes the RCBVH. But when dealing with scenes that have are very complex and contain a dense amount of triangles, there exist so many bounding volumes which do not get created needlessly. Especially since the camera almost never sees all of the triangles in the scene all at once, many bounding volumes, and therefore many triangles, get excluded from much of the ray tracing process.

The addition of concurrent implementations of the RCBVH approach also adds a new layer of both complexity and advantage. The complexity comes in the form of thread synchronization and ensuring that threads do not perform extra work and consume resources in the process and that each thread receives appropriate information so that it can operate as independently as possible. Thread synchronization adds an unavoidable overhead, one that can be seen when only using 2 threads, but as more threads are able to execute independently on CPUs and GPUs, the advantages to concurrent programming become evident very quickly.

Ray Tracing has provided the world with some of the most realistic renderings of 3D scenes either to the public via mainstream movies or scientific studies and simulations. More and more innovative techniques continue to emerge and the faster the algorithms become, the more the ray tracing community will benefit. And as the algorithms accelerate, the ability to add more realism to these renders becomes possible. One area that seems to still be untapped a bit, both in the graphics community but in general, is concurrent programming. Many programmers have begun to tap this area, so more lock-free and wait-free data-structures are emerging and can be used to aid in speeding up ray tracing. In tandem, the growth of hardware will continue to aid this process and it seems to only be getting better with time. The algorithm presented is not the end-all be-all to completely replace the state-of-the-art methods of Ray Tracing and BVH building, but can serve as a supplement to help them perform better.

Future Work

Since future CPU architectures are being geared far more towards multi-cored architectures, the author feels that future work for this algorithm should go towards the concurrent programming portion. One key point is to ensure that the threads, though operating on separate threads, do not interfere with each other's progress. To do this, the author feels that finding a way to quickly distribute the triangles such that the BVH structure built is a balanced one, would be very prudent. If the BVH tree is completely balanced, then each thread would have a much smaller chance of colliding with another. Also, studying how adding additional threads of execution past the maximum

number of threads that should be able to execute concurrently would be good. Though the threads are all running independently, there seems to have been sufficient free cycles to be able to progress the other threads, helping the concurrent execution finish a little faster. It seems that there is a point of not only diminishing returns, but also how far a shared resource be stressed. Finally, the author wishes to take a look at the N Children implementation. He felt strongly that this algorithm should have been faster than the Thread Division implementation and feels that something might not have been implemented to its fullest efficiency, slowing down the overall process. One last area of improvement would be using a much faster sorting algorithm, which may give N Children the edge of the Thread Division implementation.

APPENDIX: PROGRAM CODE

getRCBVHTrianglesAsynchronousNChildren()

```
/**
 * A method to be used to recursively build a {@link BvhTree}
 *
 * @param bvhNode
 *        The parent {@link BvhNode} to the children nodes
 * @param bvhTree
 *        The {@link BvhTree} containing the full list of children
 * @throws Exception
 */
protected void recursiveThreadedTreeBuild(ConcurrentBvhNode bvhNode,
  BvhTree bvhTree,Set<RayMap> rayMaps) throws Exception
{
  ConcurrentNChildBvhNode nChildBvhNode =
   (ConcurrentNChildBvhNode)bvhNode;

  if (rayMaps.isEmpty())
  {
    return;
  }

  Set<RayMap> intersectedRayMaps = new HashSet<RayMap>();

  for (RayMap rayMap : rayMaps)
  {
    if (VoxelUtil.rayIntersectsVoxel(rayMap.getRay(), bvhNode.getVoxel()))
    {
      intersectedRayMaps.add(rayMap);
    }
  }

  if (nChildBvhNode.getIsLeaf())
  {
    for (RayMap rayMap : intersectedRayMaps)
    {
      rayMap.getTriangles().addAll(nChildBvhNode.getTriangles());
    }
    return;
  }

  if (!intersectedRayMaps.isEmpty())
  {
    FutureTask<Boolean> buildChildrenTask = null;

    if (buildNChildren.get())
    {
      buildChildrenTask = new FutureTask<Boolean>(
        new BuildNChildrenCallable(
          nChildBvhNode, bvhTree, numThreads));
```

```
      }
      else
      {
        buildChildrenTask = new FutureTask<Boolean>(
          new BuildChildrenCallable(nChildBvhNode, bvhTree));
      }

      // Locks the node and builds children
      nChildBvhNode.lockAndExecute(buildChildrenTask);

      List<BvhNode> bvhTreeList =
       (List<BvhNode>)bvhTree.getTreeCollection();

      int sizeOfChildren = nChildBvhNode.getChildrenIndeces().size();
      int childrenProecessed = 0;

      // Checks to make sure start index is not out of bounds
      int currentIndex = nChildBvhNode.startIndex.getAndIncrement();
      while (currentIndex >= sizeOfChildren)
      {
        nChildBvhNode.startIndex.set(0);
        currentIndex = nChildBvhNode.startIndex.getAndIncrement();
      }

      while (childrenProecessed < sizeOfChildren)
      {
        recursiveThreadedTreeBuild((ConcurrentBvhNode)
          bvhTreeList.get(nChildBvhNode.getChildrenIndeces().get(
          currentIndex)), bvhTree, intersectedRayMaps);

        currentIndex++;

        currentIndex %= sizeOfChildren;
        childrenProecessed++;
      }
    }
  }


/**
 * Builds N {@link ConcurrentNChildBvhNode}s for N {@link Thread}s
 *
 * @param bvhNode
 *        The {@link ConcurrentNChildBvhNode} having children built for
 * @param bvhTree
 *        The {@link BvhTree} storing the structure
 * @param n
 *        The number of childre to build
 * @throws Exception
 */
 protected void buildNChildren(ConcurrentNChildBvhNode bvhNode, BvhTree
   bvhTree, int n) throws Exception
 {
```

```
final Plane planeOfSplit = BvhNodeUtil.getPlaneOfSplit(bvhNode);

/**
 * Comparator class
 *
 * @author kkrivera
 */
class compareTriangles implements Comparator<Triangle>
{
  @Override
  public int compare(Triangle o1, Triangle o2)
  {
    try
    {
      Vector3f centroid1 = BvhNodeUtil.getTriangleCentroid(o1);
      Vector3f centroid2 = BvhNodeUtil.getTriangleCentroid(o1);

      float val1 = centroid1.getZ();
      float val2 = centroid2.getZ();

      if (planeOfSplit == Plane.YZ)
      {
        val1 = centroid1.getX();
        val2 = centroid2.getX();
      }
      else if (planeOfSplit == Plane.XZ)
      {
        val1 = centroid1.getY();
        val2 = centroid2.getY();
      }

      if (val1 < val2)
        return 1;
    }
    catch (Exception e)
    {
      e.printStackTrace();
    }

    return 0;
  }
}

List<Triangle> triangles =
  new ArrayList<Triangle>(bvhNode.getTriangles());
Collections.sort(triangles, new compareTriangles());

bvhNode.setTriangles(new HashSet<Triangle>(triangles));

int size = triangles.size();
int div = size / n;

int start = 0;
```

```
  int end = 0;
  for (int i = 0; i < n; i++)
  {
    start = div * i;
    if (i != n - 1)
    {
      end = div * (i + 1);
    }
    else
    {
      end = size;
    }
    ConcurrentNChildBvhNode concurrentNChildBvhNode =
     (ConcurrentNChildBvhNode) buildBvhNode(new HashSet<Triangle>(
        triangles.subList(start, end)));

    bvhNode.getChildrenIndeces().add(bvhTree.getTreeCollection().size());
    bvhTree.getTreeCollection().add(concurrentNChildBvhNode);
  }

  buildNChildren.set(false);
}
```

getRCBVHTrianglesAsycnhronous()

```
/**
  * A method to be used to recursively build a {@link BvhTree}
  *
  * @param bvhNode
  *        The parent {@link BvhNode} to the children nodes
  * @param bvhTree
  *        The {@link BvhTree} containing the full list of children
  * @throws Exception
  */
  protected void recursiveThreadedTreeBuild(ConcurrentBvhNode bvhNode,
    BvhTree bvhTree,Set<RayMap> rayMaps) throws Exception
  {
    if (rayMaps.isEmpty())
    {
      return;
    }

    Set<RayMap> intersectedRayMaps = new HashSet<RayMap>();

    for (RayMap rayMap : rayMaps)
    {
      if (VoxelUtil.rayIntersectsVoxel(rayMap.getRay(), bvhNode.getVoxel()))
      {
        intersectedRayMaps.add(rayMap);
      }
    }

    if (bvhNode.getIsLeaf())
    {
      for (RayMap rayMap : intersectedRayMaps)
      {
        rayMap.getTriangles().addAll(bvhNode.getTriangles());
      }
      return;
    }

    if (!intersectedRayMaps.isEmpty())
    {
      // Locks the node and builds children
      bvhNode.lockAndExecute(new FutureTask<Boolean>(new
        BuildChildrenCallable(bvhNode, bvhTree)));

      List<BvhNode> bvhTreeList = (List<BvhNode>)
        bvhTree.getTreeCollection();

      // Determines whether the left or right side will be traversed first
      Boolean traverseLeft = null;
      do
      {
        traverseLeft = bvhNode.traverseLeft.get();
      }
```

```
  while (!bvhNode.traverseLeft.compareAndSet(traverseLeft,
    !traverseLeft));

  // Traverses the left side of the tree if traverse left is true
  if (traverseLeft)
  {
    // Traverse Left Side
    if (bvhNode.getLeftBvhNodeIndex() != null)
    {
      recursiveThreadedTreeBuild((ConcurrentBvhNode)
        bvhTreeList.get(bvhNode.getLeftBvhNodeIndex()),
        bvhTree, intersectedRayMaps);
    }

    // Traverse Right Side
    if (bvhNode.getRightBvhNodeIndex() != null)
    {
      recursiveThreadedTreeBuild((ConcurrentBvhNode)
        bvhTreeList.get(bvhNode.getRightBvhNodeIndex()),
        bvhTree, intersectedRayMaps);
    }
  }
  else
  {
      // Traverse Right Side
      if (bvhNode.getRightBvhNodeIndex() != null)
      {
        recursiveThreadedTreeBuild((ConcurrentBvhNode)
          bvhTreeList.get(bvhNode.getRightBvhNodeIndex()),
          bvhTree, intersectedRayMaps);
      }

      // Traverse Left Side
      if (bvhNode.getLeftBvhNodeIndex() != null)
      {
        recursiveThreadedTreeBuild((ConcurrentBvhNode)
          bvhTreeList.get(bvhNode.getLeftBvhNodeIndex()),
          bvhTree, intersectedRayMaps);
      }
    }
  }
}
```

getRCBVHTrianglesSynchronous()

```
/**
  * A recursive method to build the {@link BvhNode} children through
  * recursion, going down the left side of the {@link BvhTree} first
  *
  * @param bvhNode
  *        The {@link BvhNode} for which children may be created
  * @param bvhTree
  *        The {@link BvhTree} containing all of the {@link BvhNode} children
  * @throws Exception
  */
  private void recursiveTreeBuild(BvhNode bvhNode, BvhTree bvhTree,
    Set<RayMap> rayMaps) throws Exception
  {
    if (rayMaps.isEmpty())
    {
      return;
    }

    Set<RayMap> intersectedRayMaps = new HashSet<RayMap>();

    for (RayMap rayMap : rayMaps)
    {
      if (VoxelUtil.rayIntersectsVoxel(rayMap.getRay(), bvhNode.getVoxel()))
      {
        intersectedRayMaps.add(rayMap);
      }
    }

    if (!intersectedRayMaps.isEmpty())
    {
      if (bvhNode.getIsLeaf())
      {
        for (RayMap rayMap : intersectedRayMaps)
        {
          rayMap.getTriangles().addAll(bvhNode.getTriangles());
        }
        return;
      }

      if (nodeRequiresChildren(bvhNode))
      {
        buildChildren(bvhNode, bvhTree);
      }

      // Traverse Left Side
      if (bvhNode.getLeftBvhNodeIndex() != null)
      {
        recursiveTreeBuild(((List<BvhNode>)bvhTree.getTreeCollection())
          .get(bvhNode
            .getLeftBvhNodeIndex()), bvhTree, intersectedRayMaps);
      }
```

```
    // Traverse Right Side
    if (bvhNode.getRightBvhNodeIndex() != null)
    {
      recursiveTreeBuild(((List<BvhNode>)bvhTree.getTreeCollection())
        .get(bvhNode
          .getRightBvhNodeIndex()), bvhTree, intersectedRayMaps);
    }
  }
}
```

LIST OF REFERENCES

[1] Jim Hurley, Alexander Kapustin, Alexander Reshetov, Alexei Soupikov: Fast Ray Tracing on General Purpose CPUs

[2] Thiago Ize, Ingo Wald, Steven G. Parker: Asynchronous BVH Construction for Ray Tracing Dynamic Scenes on Parallel Multi-Core Architectures. In *Eurographics Symposium on Parallel Graphics and Visualization (2007)*

[3] Sung-Eui Yoon, Sean Curtis, Dinesh Manocha: Ray Tracing Dynamic Scenes using Selective Restructuring. In *Eurographics Symposium on Rendering (2007)*

[4] Lauterbach, C., Garland, M., Sengupta, S., Luebke, D. and Manocha, D. (2009), Fast BVH Construction on GPUs. *Computer Graphics Forum*, 28: 375–384.

[5] Ingo Wald, Thiago Ize, Steven G. Parker, Fast, parallel, and asynchronous construction of BVHs for ray tracing animated scenes, Computers & Graphics, Volume 32, Issue 1, February 2008, Pages 3-13, ISSN 0097-8493

[6] J. Pantaleoni and D. Luebke. 2010. HLBVH: hierarchical LBVH construction for real-time ray tracing of dynamic geometry. In *Proceedings of the Conference on High Performance Graphics*(HPG '10). Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 87-95.

[7] Kirill Garanzha, Jacopo Pantaleoni, and David McAllister. 2011. Simpler and faster HLBVH with work queues. In Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics(HPG '11), ACM, New York, NY, USA, 59-64.

[8] Ingo Wald, "Fast Construction of SAH BVHs on the Intel Many Integrated Core (MIC) Architecture," IEEE Transactions on Visualization and Computer Graphics, vol. 99, no. PrePrints, , 2010

[9] Kim, D., Heo, J.P., Huh, J., Kim, J. and Yoon, S..”HPCCD: Hybrid Parallel Continuous Collision Detection using CPUs and GPUs.” Computer Graphics Forum, 28: 1791–1800.

[10] WALD, I. 2007. “On fast construction of sah-based bounding volume hierarchies”. In *Proc. IEEE Symposium on Interactive RayTracing*, 33–40.

[11] Shirley, P., and Michael Ashikhmin. Fundamentals of Computer Graphics. Wellesley, MA: AK Peters, 2005. Print.

[12] Pharr, Matt, and Greg Humphreys. Physically Based Rendering from Theory to Implementation. San Francisco, CA: Morgan Kaufmann, 2010. Print.